View and Verify Access Control Policies

João Sá

August 5, 2016

To Adriana.

Acknowledegment

I would like to express my gratitude to Professors Sandra Alves and Sabine Broda for their guidance on this work, along with their advice and comments on this thesis.

My sincere appreciation to my employer, Alert Life Sciences Computing, for allowing me to dedicate part of my working hours to this master.

I would like to thank my mother, sister and brother for their constant support.

Summary

This thesis describes the development of a prototype, the G-ACM (Graphical Access Control Manager), which provides a graphical interface for visualization and management of access control policies.

The main objective of this work is to explore the advantages of the graphical representation to assist users with the tasks of management of security policies.

The prototype consists of two modules. The server component uses the Drools rule engine to implement the semantic of the Category Based Access Control (CBAC) model, which generalizes several well known access controls models, such as Mandatory Access Control (MAC), Discretionary Access Control (DAC) and Role Based Access Control (RBAC). The client, the G-ACM Console, uses the graphical representation of access control policies to provide an user friendly environment to describe and manage CBAC policies.

G-ACM exposes some of the advantages of the graphical representation of policies when compared to more traditional representations. For a more extensive analysis some important features should be added to the prototype. Nevertheless, from this work we may conclude that the exploration of the type of representation used by G-ACM is a path worth exploring.

Sumário

Esta tese descreve o desenvolvimento de um protótipo, o G-ACM (Graphical Access Control Manager), que disponibiliza um interface gráfica para visualização e gestão de políticas de controlo de acessos.

Este trabalho tem como principal objectivo explorar as vantagens da representação gráfica para apoiar os utilizadores nas tarefas de administração das políticas de segurança.

Este protótipo é composto por dois módulos. A componente servidora usa o motor de regras Drools para implementar a semântica do modelo Category Based Access Control (CBAC) que generaliza alguns modelos amplamente conhecidos como os Mandatory Access Control (MAC), Discretionary Access Control (DAC) e Role Based Access Control (RBAC). A componente cliente, o G-ACM Console, usa a representação gráfica das políticas de controlo de acessos para providenciar um ambiente de utilização amigável para descrever e gerir políticas CBAC.

O G-ACM expõem algumas das vantagens da representação gráfica das políticas de segurança quando comparada com representações mais tradicionais. Para elaborar uma análise mais extensiva algumas funcionalidades importantes deveriam ser incluídas no protótipo. No entanto, este trabalho permite concluir que a exploração do tipo de representação usado no G-ACM é um caminho que vale a pena explorar.

Contents

List of Figures

Acronyms

BRMS Business Rules Management System.

BTG Break The Glass.

CBAC Category-Based Access Control.

DI Dependency Injection.

DOM Document Object Model.

DTO Data Transfer Object.

G-ACM Graphical Access Control Manager.

HTML HyperText Markup Language.

HTTP Hypertext Transfer Protocol.

IoC Inversion of control.

MLS Multi Level Security.

MVC Model-view-controller.

ORBAC Organisation Based Access Control.

RBAC Role Based Access Control.

W3C World Wide Web Consortium.

Chapter 1

Introduction

Access control aims to restrict the access to information, or other system's resources, to authorized people or computer programs. The increasing number of devices connected to the internet and the use of cloud systems demand a revision of the security properties and mechanisms used to provide security [1]. The importance of access control enforcement increases as data and computation become more distributed, becoming a critical issue nowadays.

Authorization is the process of verifying if a *principal* (e.g. a user) has access to perform a requested operation. It relies on a previous authentication step, where the identity of the principal is verified. In this work we focus on authorization and assume that principals are authenticated beforehand. Authorization consists of two phases. The first is the definition of the authorization policy that specifies if, and under which circumstances, principals have access to resources. The second is the process of enforcing the defined policies during system operation.

More precisely, policies specify if principals are allowed to perform actions on resources. Resources are any type of asset on which actions are performed (e.g. printers, database records, systems' features, etc). Principals are the subject performing the action and can have many forms (e.g. users, programs, systems, etc). A pair of an action and a resource is called a permission.

Maintaining authorization data is not trivial. Policies can easily get complex and it is often necessary to change access rules in the system. This task is increasingly difficult in distributed systems, which makes the development of tools, to help defining and

managing authentication policies, an ever more important task.

Several access control models are in use nowadays, some common examples are:

- *Mandatory Access Control (MAC)*: usually associated with the Bell-LaPadula model of Multi Level Security (MLS) [2], can be defined as any access control model that enforces security policies independently of user operation. This model labels resources by their sensitivity and defines authorization levels (or clearance) for principals. Permissions are enforced by comparing labels with authorization levels. This method is broadly known by its use in the military context, but the concept was also implemented in several commercial products (e.g. Oracle Label Security in Oracle RDBMS).

- *Discretionary Access Control (DAC)*: associates to each user a list of permissions and prohibitions. It is widely used in file systems of common operating systems. It was developed to implement Access Control Matrices defined by Lampson in his paper on system protection [3]. It allows fine-grained control over system objects which allows implementing least-privilege access. It is intuitive and mostly invisible to users which makes it to be regarded cost-effective for small systems [4].

- *Role Based Access Control (RBAC)*: first proposed by Ferraiolo and Kuhn in 1992, the *RBAC* model is based on the notion of *role* to which permissions are mapped. Principals get their permissions according to the roles they are assigned to. Permission mapping is controlled by a security administrator, therefore users cannot transfer their permissions to other users, which make it behave as a finer-grained version of the Mandatory Access Control (MAC) model. Several *RBAC* variations were introduced by [5] and adopted as ANSI standards. One example is the enforcement of separation of duties by the addition of constraints for a user to enter a role. Another is the introduction of hierarchies to define inheritance relations among roles, which reduce administration costs (Hierarchical RBAC [5]).

Even though widely used, these models have some inherent limitations.

MAC model assumes that the policies do not change (the tranquility principle), which prevents the dynamic alteration of the permissions. Changing the classification of resources or subjects requires the use of "trusted subjects", which imply that large parts of the operating system and utilities must be placed outside the access control

framework. Additionally, the model can hurt productivity by over-classifying data and it does not assures fine-grained least privilege.

Using Discretionary Access Control (DAC), the verification of permissions and the maintenance of systems using it is extremely difficult. Additionally, since permissions are controlled by users, it allows potential exploits by Trojan horses.

Role Based Access Control (RBAC) addresses most of the limitations of DAC. It removes the control of permissions from the users, preventing the transfer of permissions to other users. Additionally, by assigning permissions to roles, it simplifies the task of managing permissions. Nevertheless, maintaining permission for large systems can still be very changeling. Administration tasks, like role membership, role inheritance and finer-grained customized privileges can become rapidly unmanageable for large number of roles and resources.

In [6], the Category-Based Access Control (CBAC) model was proposed as a formalism from which the other models can be derived. The motivation was to direct the research efforts to a unifying metamodel, rather than developing new ones for specific purposes and with limited reach.

CBAC brings together the set of core principles of access control and simplifies the task of access control reasoning by removing the specificities of the different models. A rewrite-based operational semantics for the model is defined in [7], which is used to extract properties on policies, such as totality and consistency. The expressive power of the model was illustrated by deriving the models mentioned above as instances of CBAC.

The notion of *category* is central to the model, as permissions are assigned to categories of principals and not to individual principals. Categories define classes of entities that share some property, such as an user attribute, a time or geographical constraint or a resource property. This implies that a principal's permission may change dynamically due to changes of the values of these attributes. For instance, suppose an online voting system that verifies if a person is allowed to vote. It defines a category *Adult* and all users assigned to that category have permission to cast votes. Whenever a person reaches the age of eighteen, he/she is automatically mapped to that category and becomes automatically able to vote.

The graphical representation of the category based model in [8, 9] can be used to simplify policy administration and verification tasks. The potential of the framework

is illustrated by using it to deal with emergency situations in an hospital environment, where the global policy results from the dynamic combination of the normal policy with a specific one for emergency contexts.

In this work we develop a prototype, called G-ACM (Graphical Access Control Manager), that implements the semantics of the model and provides a graphical interface for visualization and management of access control policies. The tool uses the graphical representation of the CBAC model introduced in [8], and further developed in [9], to provide a user friendly environment to describe and manage CBAC access control policies.

This prototype has two main components. The first is a server that uses the Drools rule engine to implement the semantics of CBAC with dynamic policies. The second is a client application that provides the representation of the policies as graphs and allows the simulation of authorization scenarios by changing the parameters used by the policies.

The long term goal of this work is to evaluate the usefulness of the graphical representation to assist users on the management of security policies. We expect G-ACM to be used for this type of analysis. Even though we do not perform a formal comparison between this type of representation and the traditional one, we briefly discuss some obvious advantages of the graphical representation that can be further explored.

Overview The rest of this thesis is organized as follows. In the Related Work (Chapter 2) we briefly introduce other works that explore the use of graphs to represent security policies. In Chapter 3 we describe the CBAC model, the graphical representation of policies, and present a set of authorization use cases that provide a concrete context. In Chapter 4 we briefly describe the Drools rule engine and how it is used to implement the semantics of the model. In Chapter 5 we describe the G-ACM from the conceptual point of view, and in Chapter 6 its technical implementation. Finally, in Chapter 7 we draw some conclusions and discuss future work.

Chapter 2

Related Work

This work is based on the representation of CBAC policies introduced in [8] and further developed in [9]. Within CBAC, only textual languages have been used and have focused mainly on the expressivity of the model, the analysis of policies, and the techniques that can be used to enforce policies [6, 7, 10, 11, 12].

Several other access control models have been studied through the use of graph-based languages. For example, Koch and al. [13, 14] use directed graphs to formalize RBAC. A distinctive feature in this work is the use graph transformation rules to model role management operations. The graphs in [13, 14] and [9] are both typed and labelled. The typing system is similar in both cases but the label structure in [9] is richer so it can express policies where access rights depend on data associated to the entities in the policy. Labels in [13, 14] are simply identifiers used to encode RBAC.

The RBAC policies in [13, 14] can be represented by graphs in [9], since a role is a particular case of a category. However, graphs used in [13, 14] represent also session information, which is not dealt by policy graphs in [9]. Nevertheless, since the notion of session in RBAC is similar to the same notion in CBAC, the representation of sessions provided in [13, 14] could be easily adapted to policy graphs representing CBAC policies.

LaSCO [15] represents policies through graphs in which nodes represent system objects and edges represent system events. In LaSCO, each policy graph represents both the situation under which a policy applies and the constraint that must hold for the policy to be upheld. As discussed in [9], the model proposed by Koch and al., LaSCO and

other models [16, 17, 5], can be represented by CBAC policies.

Another approach consists in using term rewriting systems to express general dynamic access control policies [18, 19, 20, 21, 11]. Term rewrite rules describe particular access control models, which can be used to verify the properties of policies by checking the confluence and termination of sets of rewrite rules. The approach in [9] uses rewrite rules to model the dynamics of the system and a visual graph formalism to represent a concrete state of the system. In [20], it is shown how narrowing can be used to solve administrator queries of the form "what if a request is made under these conditions?", by representing a query as a pair of a term and a first-order equational constraint. Narrowing-based techniques could also be used in dynamic graph policies, since the functions defining the main relevant relations are defined by sets of rewrite rules. The extensive theory of rewriting provides an strong platform to establish security properties [19, 22, 23]. Additionally, it allows using rewriting-based frameworks (such as CiME, MAUDE or TOM) to reason about those properties.The work in [9] addresses similar issues, but is based on a notion of category-based access control for distributed environments, which uses labelled graphs to include concepts like time, events, and histories that are not included as elements of RT or RBAC. In [24], CiME is integrated in a tool designed to automatically check consistency and totality of RBAC access control policies. A similar technique could be used to analyse the rewrite system in a dynamic policy graph.

Several extensions of RBAC, which deal with dynamic permissions, have been proposed. These models allow permissions to change according to internal or external conditions such as time, location, or context-based properties (see, for example, [25, 26, 27]). All these extensions can be modeled as instances of CBAC.

Even though graphs are used in several models to represent and verify the properties of access control policies, there is almost no literature on tools that take advantage of the visual representation of graphs to manage the policies. The only exception that we found is the *Policy Manager* described in [28]. The *Policy Manager* implements a user friendly representation of policies similar to the representation used in the G-ACM. On the other hand, in the *Policy Manager* [28], the dynamic behaviour of CBAC policies is achieved through the edition of Ruby code, which requires the users to have knowledge of the language.

Most tools dealing with management of access control policies use tables to represent permissions data. For instance, the MotOrBAC tool [29] allows to specify and simulate

policies using the Organisation Based Access Control (ORBAC) model. The ORBAC model [30] defines security policies centered on the organisation. Besides permissions, the model supports prohibitions and obligations. To address dynamism, the model uses *contexts*, which express the conditions under which permissions are active. It includes a conflict detection feature to assist the user at finding and solving conflicts. Through its interface the user can perform all tasks related to the creation of policies and corresponding entities, which makes it very complete. However, MotOrBAC's usability has some limitations, due to the tabular representation of permissions and conflicts, which is hard to read. We believe that the usability of this kind of tool would benefit greatly from the inclusion of a graphical representation of policies.

Chapter 3

Preliminaries

This chapter introduces the concepts underlying the development of the G-ACM tool. Section 3.1 describes CBAC, the category-based model. Section 3.2 introduces the graph representation of CBAC policies defined in [8] and later developed in [9]. Finally, Section 3.3 describes the set of authorization scenarios that are used in the examples throughout the rest of this thesis.

3.1 The Category-Based Model

The following description introduces the key concepts of the category-based model (for more details we refer to [6]).

We consider the following sets of entities: a countable set $\mathcal{C}$ of *categories*, a countable set $\mathcal{P}$ of *principals*, a countable set $\mathcal{A}$ of named *actions*, a countable set $\mathcal{R}$ of *resource identifiers*, a finite set $\mathcal{AUTH}$ of possible *answers* to access requests (e.g., {grant, deny, undetermined}) and a countable set $\mathcal{S}$ of situational identifiers to denote context data.

The model defines the following relations:

- *Principal-category assignment:* $\mathcal{PCA} \subseteq \mathcal{P} \times \mathcal{C}$, such that $(p, c) \in \mathcal{PCA}$ iff a principal $p \in \mathcal{P}$ is assigned to the category $c \in \mathcal{C}$.
- *Permission-category assignment:* $\mathcal{ARCA} \subseteq \mathcal{A} \times \mathcal{R} \times \mathcal{C}$, such that $(a, r, c) \in \mathcal{ARCA}$ iff the action $a \in \mathcal{A}$ on resource $r \in \mathcal{R}$ can be performed by the principals assigned

to the category $c \in \mathcal{C}$.

- *Authorisations:* $\mathcal{PAR} \subseteq \mathcal{P} \times \mathcal{A} \times \mathcal{R}$, such that $(p, a, r) \in \mathcal{PAR}$ iff a principal $p \in \mathcal{P}$ can perform the action $a \in \mathcal{A}$ on the resource $r \in \mathcal{R}$.

Definition 3.1 (Axioms). The relation $\mathcal{PAR}$ satisfies the following core axiom, where we assume that there exists a relationship $\subseteq$ between categories; this can simply be equality, set inclusion or a specific relation may be used.

$$\textit{(a1)} \quad \forall p \in \mathcal{P},\ \forall a \in \mathcal{A},\ \forall r \in \mathcal{R}, \quad (p, a, r) \in \mathcal{PAR} \Leftrightarrow$$
$$\exists c, c' \in \mathcal{C}, ((p, c) \in \mathcal{PCA} \wedge\ c \subseteq c'\ \wedge (a, r, c') \in \mathcal{ARCA})$$

Axiom *(a1)* states that a request by a principal p to perform the action a on a resource r is authorised only if p belongs to a category c such that for some category c' such that $c \subseteq c'$ (e.g., c itself), the action a is authorised on r, otherwise the request is denied. There are other alternatives, e.g., considering *undeterminate* as answer if there is not enough information to grant the request. Operationally, Axiom *(a1)* can be realised through the following rewrite-based specification:

$$par(P, A, R) \rightarrow\ if (A, R) \in arca^*(contain(pca(P)))\ then\ grant\ else\ deny \tag{3.1}$$

Function *par(P,A,R)* above relies on functions *pca*, which returns the list of categories assigned to a principal, and *arca*, which returns a list of permissions assigned to a category. Function *contain* computes the set of categories that contain (w.r.t. relation $\subseteq$) any category returned by *pca(P)*. The function $\in$ is a membership operator on lists, *grant* and *deny* are answers, and $arca^*$ generalises the function *arca* to take into account lists of categories:

To deal with prohibitions, relations *Banned actions on resources* ($\mathcal{BARCA}$) and *Prohibitions* ($\mathcal{BAR}$), can be defined. Their definitions are equivalent to those of $\mathcal{ARCA}$ and $\mathcal{PAR}$ respectively, but instead of defining authorizations they define prohibitions.

- *Banned actions on resources:* $\mathcal{BARCA} \subseteq \mathcal{A} \times \mathcal{R} \times \mathcal{C}$, such that $(a, r, c) \in \mathcal{BARCA}$ iff the action $a \in \mathcal{A}$ on resource $r \in \mathcal{R}$ is forbidden for principals assigned to the category $c \in \mathcal{C}$.

- *Banned access:* $\mathcal{BAR} \subseteq \mathcal{P} \times \mathcal{A} \times \mathcal{R}$, such that $(p, a, r) \in \mathcal{BAR}$ iff performing the action $a \in \mathcal{A}$ on the resource $r \in \mathcal{R}$ is forbidden for the principal $p \in \mathcal{P}$.

Definition 3.2 (Axioms). The relation $\mathcal{BAR}$ satisfies the following core axioms:

(a2) $\forall p \in \mathcal{P},\ \forall a \in \mathcal{A},\ \forall r \in \mathcal{R},\ (p,a,r) \in \mathcal{BAR} \Leftrightarrow$
$\exists c, c' \in \mathcal{C}, ((p,c) \in \mathcal{PCA} \wedge\ c \subseteq c'\ \wedge (a,r,c') \in \mathcal{BARCA})$

The inclusion of $\mathcal{BARCA}$ has yet other implications. For access requests that are neither authorized nor denied, a function $\mathcal{UNDET}$ may be defined. Furthermore, it must be assured that the same request is not simultaneously authorized and denied. Therefore, if the relation $\mathcal{BARCA}$ is admitted then the following additional axioms are needed:

(a3) $\forall a \in \mathcal{A},\ \forall r \in \mathcal{R},\ \forall s \in \mathcal{S}$
$((p,a,r) \notin \mathcal{PAR} \wedge (p,a,r) \notin \mathcal{BAR}) \quad \Leftrightarrow (p,a,r) \in \mathcal{UNDET}$

(a4) $\mathcal{PAR} \cap \mathcal{BAR} = \varnothing$

The rewrite-based specification of Axioms *(a1)*, *(a2)*, *(a3)* and *(a4)* is given by the rewrite rule:

$$\begin{aligned} par(P,A,R) \quad \rightarrow\ & if(A,R) \in arca^*(contain(pca(P)))\ then\ grant \\ & else\ if(A,R) \in barca^*(isContained(pca(P)))\ then\ deny \\ & else\ undeterminate \end{aligned} \tag{3.2}$$

Function *isContained* computes the set of categories that are contained (w.r.t. relation $\subseteq$) by any category returned by *pca(P)*.

Definition 3.3 (Category-based policy with prohibitions). A CBAC policy with prohibitions is a tuple $\langle \mathcal{E}, \mathcal{PCA}, \mathcal{ARCA}, \mathcal{PAR}, \mathcal{BARCA}, \mathcal{BAR} \rangle$, where $\mathcal{E} = (\mathcal{P}, \mathcal{C}, \mathcal{A}, \mathcal{R}, \mathcal{S})$, such that Axioms *(a1)*, *(a2)*, *(a3)* and *(a4)* are satisfied.

Note that, provided that the axioms are satisfied, relations $\mathcal{PCA}$, $\mathcal{ARCA}$ and $\mathcal{BARCA}$ can evolve in time. This means that, as the system state changes, so do allocation of principals to categories and the mapping between categories and authorizations and prohibitions.

A distributed version of the category-based model was proposed in [10]. It describes a scenario in which several sites define their own policies, described by local $\mathcal{PCA}_s$,

$\mathcal{ARCA}_s$, $\mathcal{BARCA}_s$, $\mathcal{PAR}_s$ and $\mathcal{BAR}_s$, that contribute to a global authorization. Policies in each site must satisfy the axioms described above as well as the following:

(b1) $\forall p \in \mathcal{P},\ \forall a \in \mathcal{A},\ \forall r \in \mathcal{R},$
$(p, a, r) \in\ \mathcal{OP}_{par}(\{\mathcal{PAR}_s, \mathcal{BAR}_s | s \in S\})\mathcal{PAR} \quad \Leftrightarrow (p, a, r) \in \mathcal{PAR}$

(b2) $\forall p \in \mathcal{P},\ \forall a \in \mathcal{A},\ \forall r \in \mathcal{R},$
$(p, a, r) \in\ \mathcal{OP}_{bar}(\{\mathcal{PAR}_s, \mathcal{BAR}_s | s \in S\})\mathcal{BAR} \quad \Leftrightarrow (p, a, r) \in \mathcal{PAR}$

(b3) $\mathcal{PAR} \cap \mathcal{BAR} = \varnothing$

Axioms *(b1)* and *(b2)* describe the global operators $\mathcal{OP}_{par}$ and $\mathcal{OP}_{bar}$ that compute the final authorization from the authorizations in each site. These operators are application specific and can be very simple, for instance, only allow access requests if the answer is *allow* for all sites, or can be more complex to fulfill specific application needs.

3.2 Graph Representation

In the graph representation of CBAC policies, defined in [8] and later developed in [9], policies are represented through *labelled graphs* where nodes represent entities and paths represent relations.

Data is attached to nodes and edges through *labels*. For example, in [9] labels are records that store data in the form of key-value pairs, where all records have an attribute *ent* that identifies the entity it belongs to. In the following definitions we assume the existence of a proper representation for labels. We will use the term *attribute* to refer to the data stored in the labels of nodes and edges.

To represent policies without prohibitions (i.e., satisfying Axiom *(a1)*) the following definition is given:

Definition 3.4 (Policy graph). A policy graph is a tuple $\mathcal{G} = (V, E, lv, le)$, where $\mathcal{V}$ is a set of nodes, E is a set of undirected edges $\{v_1, v_2\}$, where $v_1, v_2 \in \mathcal{V}$ and $v_1 \neq v_2$, lv is a labelling function for nodes, and le is a labelling function for edges.

Attribute *type* identifies the type of entity that the node represents (*P* for principal, *C* for category, *A* for action and *R* for resource). The type of an edge derives from the type of its adjacent nodes. For instance, an assignment of a principal to a category is represented by an edge of type *PC*. More precisely, if $e = \{v_1, v_2\}$ then the type of edge *e* is the pair formed by the types T_1, T_2 of nodes v_1, v_2, respectively and it is represented by T_1T_2.

The relation $\mathcal{PAR}$ can be computed from the *paths* in a graph, which are defined as follows:

Definition 3.5. A *path* in $\mathcal{G}$ of length *n*, between two nodes v_0, v_n, is a sequence $v_0, v_1, ..., v_n$, such that, for all $1 \leq i \leq n, \{v_{i-1}, v_i\} \in E$.

Additionally, to define paths that take into account the relation $\subseteq$ between categories an additional attribute *target* on edges is needed. It defines the direction in which *CC* edges can be traversed. More precisely if v_i belongs to the target of e (notation: $v_i \in target(e)$) then v_i is a destination node of edge e.

Furthermore, the type of a *CC* edge $e = \{v_1, v_2\}$ is denoted by $\overrightarrow{CC}$ if $v_2 \in target(e)$, and by $\overleftarrow{CC}$ if $v_1 \in target(e)$. That is, type $\overrightarrow{CC}$ means that the edge is traversed from a category c_1 to a category c_2 where $c_1 \subseteq c_2$, that is, from a more specialized to a broader category, and type $\overleftarrow{CC}$ means that the edge is traversed in the opposite direction.

It is then possible to define a well formed policy graph and characterize its relations in terms of paths.

Definition 3.6 (Well-formed policy graph). A policy graph is well formed if it contains only edges of type PC, CC, CA and AR.

Proposition 3.7. In a well-formed policy graph, paths starting in a node of type P and ending in a node of type R must start with an edge of type PC, followed by edges of type CC, and finally edges of type CA and AR. Any path of size 3 must have the following shape:

Using attribute *target*, it is possible to give a definition of path that expresses the hierarchical relation between categories:

Definition 3.8. A *constrained path* of length *n* in a graph $\mathcal{G}$, between two nodes v_0, v_n, is a sequence $v_0, e_1, v_1, e_2 ..., e_n, v_n$, such that for every *CC* edge $e_i = \{v_{i-1}, v_i\}$ one has $v_i \in target(e_i)$, $i \in \{1, \ldots, n\}$.

Types of paths are defined from the types of their edges:

Definition 3.9 (Types for paths). Let $v_0, v_1, ..., v_n$ be a path of length n, such that T_i is the type of v_i for $0 \leq i \leq n$. The type of the path is the sequence given by the types of the edges along the path, that is $T_0T_1, T_1T_2, ..., T_{n-1}T_n$. The notation $type(v_0, v_1, ..., v_n) = T_0T_1, T_1T_2, ..., T_{n-1}T_n$ will be used to indicate that there is a path $v_0, v_1, ..., v_n$ and its edges have types $T_0T_1, T_1T_2, ..., T_{n-1}T_n$.

The following image shows a constrained path of type PC,$\overrightarrow{CC}$,$\overrightarrow{CC}$,CA,AR:

As a consequence of Definitions 3.8 and 3.9, the relation $c_1 \subseteq c_2$ between categories c_1 and c_2 is represented in the policy graph by a path of type $(\overrightarrow{CC})^*$. The relations $\mathcal{PCA}_\mathcal{G}$, $\mathcal{ARCA}_\mathcal{G}$ and $\mathcal{PAR}_\mathcal{G}$ are defined in terms of paths of type PC, of type CA, AR, and of type $PC, (\overrightarrow{CC})^*, CA, AR$, respectively.

From the above, it follows that a well-formed policy graph defines a unique CBAC policy and that for each policy there exists a well-formed policy graph that represents it.

To represent CBAC policies with prohibitions, relations $\mathcal{BARCA}_\mathcal{G}$ and $\mathcal{BAR}_\mathcal{G}$ must also be represented. This is achieved by using an extra attribute *auth* on edges *CA*, with possible values *A,B* to represent authorization and prohibitions (banned actions). The type of edges that represents authorizations is denoted by CA^A and the type that represents prohibitions is denoted by CA^B.

To accommodate prohibitions the definition of well-formed graph is adapted as follows:

Definition 3.10 (Well-formed policy graph with prohibitions). A policy graph with prohibitions is well formed if it contains only edges of type PC, CC, CA^A, CA^B and AR.

Computing $\mathcal{BAR}_{\mathcal{G}}$ involves traversing *CC* edges in the inverse direction, since prohibitions are inherited from more specialized to broader categories. To do so, we consider constrained inverse paths between nodes of type *C*, which are characterized by type $(\overleftarrow{CC})^*$.

In a well formed policy graph with prohibitions, relations $\mathcal{PCA}_{\mathcal{G}}$, $\mathcal{ARCA}_{\mathcal{G}}$, $\mathcal{BARCA}_{\mathcal{G}}$, $\mathcal{PAR}_{\mathcal{G}}$ and $\mathcal{BAR}_{\mathcal{G}}$ are defined in terms of paths of type PC, of type CA^A, AR, of type CA^B, AR, of type $PC, (\overrightarrow{CC})^*, CA^A, AR$, and of type $PC, (\overleftarrow{CC})^*, CA^B, AR$, respectively.

As before, it follows that a well-formed policy graph with prohibitions defines a unique CBAC policy with prohibitions and that for each policy there exists a well-formed policy graph with prohibitions that represents it.

A *distributed policy graph* is also defined in [8] and [9]. This graph is defined through the extension of the well-defined policy graph with prohibitions with the inclusion of site location identifiers in the labels.

The distributed policy graph is a tuple of graphs $(\mathcal{G}_{s1}, ..., \mathcal{G}_{sn}, \mathcal{OP})$ where $\mathcal{OP}$ is the operation on graphs that will be used to defined the global policy. For each location $s \in S$ the relations $\mathcal{PCA}_s$, $\mathcal{ARCA}_s$, $\mathcal{BARCA}_s$, $\mathcal{PAR}_s$ and $\mathcal{BAR}_s$ are defined as in the non-distributed scenario.

To represent the dynamics that exist in CBAC policies, a *dynamic policy graph* is defined in [9] as a well formed policy graph for which a function is defined that, given a principal, returns the list of categories it is assigned to and, given a category, returns the assigned lists of authorizations and prohibitions. This definition encompasses the idea that changes in the system may lead to changes in assignment of entities. Hence, a specific graph provides relations $\mathcal{PCA}$, $\mathcal{ARCA}$, $\mathcal{BARCA}$, $\mathcal{PAR}$ and $\mathcal{BAR}$ for a particular system state.

3.3 Use Cases

The following real world scenarios, gathered in an early stage of this work, are related to the access to patient health records. Due to the complex and dynamic nature of clinical environments, they provide many different use cases of access control. Simultaneously, we tried to choose scenarios that represent general problems that, with the proper adaptations, can be easily translated to other systems and environments.

All the examples in this thesis and implemented in the prototype application are based in the following use cases:

- *System's actions per role*: the actions users can perform in the system depend on their professional role. For instance the set of actions physicians and nurses can perform are different. Access control policies should be able to define which users have access to each system actions, according to their role.
- *User action allowed/denied by senior staff*: senior clinicians can give/take access to specific actions to the professionals under their supervision (either by adding/removing them to/from categories or by adding/removing permissions from the categories they belong to).
- *Clinical record access for non responsible clinicians*: clinical records are accessible to the group of physicians and nurses involved in a patient's treatment. When a clinician tries to access a record of a patient that is not under his care, the system asks the user to insert a valid reason to access the record. This kind of policy is called Break The Glass (BTG) [31, 32], this name derives from the process of breaking the glass to pull a fire alarm. In [33], the integration of the BTG concept in the standard RBAC is proposed. The resulting model is called BTG-RBAC.
- *Critical state*: under normal circumstances clinicians can only access the records of the patients under their care, but if the patient status is set to critical then all clinicians can access his records.
- *Seal, Seal and Lock policies*: in UK's National Health System (NHS), besides the restrictions described in the previous points, patients can request additional restrictions:
  - *Seal:* A patient can *Seal* the parts of his record that he/she feels are more sensitive. Access to those parts of the record requires an additional justification by the user (and corresponding log and alerts).
  - *Seal and Lock* is a stronger version of *Seal*; in this case the record will be totally inaccessible except for its author. We refer to [34] "HSCOC Information governance. Patient choices " for further details.

A key aspect of the CBAC model described previously is the ability of authorizations to change dynamically due to the possibility of defining categories based on system's

attributes. All scenarios above, except the first, which can be dealt with a traditional RBAC, imply some level of adaptability based on the patient status, its relation with the user, or any other attributes in the system.

Chapter 4

A Drools Rule Engine for CBAC

This chapter describes the use of the Drools rule engine in the implementation of the G-ACM tool. Section 4.1 provides a brief description of the Drools rule engine and Section 4.2 describes the use of the Drools rule engine to compute dynamic CBAC policy graphs.

As described in [9], the relation between entities in a dynamic CBAC policy graph changes autonomously as a result of events in the system. A particular graph, or the set of authorizations ($\mathcal{PAR}$ and $\mathcal{BAR}$) that it expresses, represents a particular system state.

In practical terms, this imposes some challenges regarding authorization verification and maintenance:

- The policy graph must be computed for every access request.
- The set of functions that, from the events in the system, compute the mappings principals/categories and principals/permissions are implementation specific and may change frequently to accommodate environmental changes.
- To manage authorizations, traditional tools are not enough. A security administrator needs to be able to verify how changes in the system's state affect permissions.

These restrictions were at the base of the decision to use a Business Rules Management System (BRMS) to describe and compute authorizations. We chose JBoss Drools

[35], because it is open source, well documented and under active development.

Drools simplifies the task of writing and updating the rules for a specific implementation. Its engine provides a simple API which allows updating the rules evaluation context to match the system's state. In the next section we describe briefly the Drools rule engine and in the subsequent section we describe how we use Drools rules to compute dynamic CBAC policy graphs.

4.1 Drools

The following description introduces the main concepts of the Drools rule engine (see [35] for the engine's full documentation).

The Drools rule engine is based on the Rete pattern matching algorithm [36] for production rule systems, which implements forward chaining using directed acyclic graphs to represent rules. Forward chaining is a reasoning technique that applies inference rules to the available data to extract more data. The inference engine iterates through this process until a goal is reached.

We start by describing the Drools native rule language. Very briefly, a rule is a declaration of the form:

```
rule "name"
   Attributes
when
   LHS
then
   RHS
end
```

where *name* is the unique identifier of the rule, `Attributes` is a list of optional features that can influence the behaviour of the rule, `LHS` (Left Hand Side) specifies a particular set of conditions, and `RHS` (Right Hand Side) is a block of code specifying the actions to be executed, should the conditions in `LHS` be satisfied.

The list `Attributes` is a (possibly empty) sequence of attributes from the following set: `no-loop`, `ruleflow-group`, `lock-on-active`, `salience`, `dialect`, `agenda-group`, `auto-focus`, `activation-group`, `date-effective`, `date-expires`, `duration`.

We will use attributes `dialect` and `salience`. Attribute `dialect` specifies the language in use in the `LHS` expressions or the `RHS` code, which currently can be Java or MVEL. The default value is the one specified at the package level, but this attribute allows this to be overridden for a particular rule. Attribute `salience` allows to define priority between rules. The default value for salience is zero, but rules with higher salience have priority over rules with lower salience. This attribute can also be defined dynamically using bound variables.

`LHS` consists of zero or more *Conditional Elements*, which determine when the rule applies. If `LHS` is empty, then the conditional element is considered to be true, causing the rule to be activated once when a new session is created (in Drools, rules are evaluated in the context of a *session* into which data can be inserted and from which process instances can be created). Conditional elements work with patterns, which can match facts currently in the working memory. Patterns can contain zero or more constraints. Because of the amplitude of options, we do not describe the full syntax for conditional elements. We refer the reader to [35], for the full description.

The `RHS` represents the action part of the rule, whose main objective is to insert, modify or delete data in the working memory. This part of the rule corresponds to code in a supported dialect. It should be atomic and not contain conditional/imperative code, since it represents what to do when `LHS` is valid. There are methods available for conveniently changing facts in memory, such as: `update(object, handle)`, `update(object)`, `insert(new Object())`, `delete( handle)`, etc. Again, we refer to [35] for all available methods.

As stated before, rules are evaluated within a session, which can be of two types: *stateless* or *stateful*. In stateless sessions, changes to the inserted facts (in the `RHS`) are not available to the rule engine. Stateful sessions, on the other hand, allow *inference*, i.e., whenever a fact is changed all the rules that use that fact are reactivated. The rules are called iteratively until there are no more changes.

The G-ACM uses a *stateful* session, which is fed with the rules and two other kinds of data:

Globals: Globals are objects that are made visible to the rule engine but changes in them do not trigger reevaluation of rules. They provide context information that can be used to evaluate rules. Furthermore, they are used as a vehicle for returning results

from a session.

Facts: Facts are the data objects used by rules when evaluating the `LHS`. The set of facts in one session provides the evaluation context for the rules. This means that, to compute authorizations for a particular system state, the session should be fed with a set of facts that corresponds to the system events that influence rule evaluation.

For instance, use case *User action allowed/denied by senior staff* in Section 3.3 describes a scenario where some users can link/unlink other users to categories. The way a particular system implements that capacity is out of the scope of this work, but we can assume that the given permission is recorded somehow in the system. To compute the set of authorizations, an object (or fact that corresponds to that information), should be inserted in the rules session prior to computation. The rule that implements this logic is as follows:

```
rule "Rule add customs Pcas"
    when
     $pca : SetPca()
    then
     insert(new Pca($pca.getPrincipal(), $pca.getCategory()));
end
```

Object `SetPca` is the fact that corresponds to the explicit mapping between principals and categories in the system. Since its existence is the only restriction on the above condition, then, for each `SetPca` fact in the session, a new `Pca` fact will be created and it will have the same values for its attributes `Principal` and `Category` as `SetPca`.

In the next section we provide further examples on how the use cases were translated into rules and we explain how the final set of authorizations is computed.

4.2 Rule description

As mentioned in the previous section, to compute the relations $\mathcal{PCA}$, $\mathcal{ARCA}$ and $\mathcal{BARCA}$ (and corresponding $\mathcal{PAR}$ and $\mathcal{BAR}$), the Drools session uses the objects that represent the system state. We will refer to those objects as *custom facts*.

Besides the custom facts, the session also needs CBAC's base entities, *principal*, *category*, *action* and *resource*. We designate this group of entities as *base entities*.

Furthermore, we may consider an initial set of relations $\mathcal{PCA}$, $\mathcal{ARCA}$ and $\mathcal{BARCA}$ (given by an RBAC system, for instance) that provides the baseline for authorizations (these would be enough to satisfy use case *System's actions per role* in Section 3.3). We call this group *base relations*.

From the point of view of a Drools session, all these types of entities are *facts* because they are all involved in the evaluation of rules conditions.

Computing the sets of authorizations, $\mathcal{PAR}$ and $\mathcal{BAR}$, consists of two distinct phases. In the first, the sets of relations $\mathcal{PCA}$, $\mathcal{ARCA}$ and $\mathcal{BARCA}$ are computed from the facts. During the second phase, the rewriting rule (3.2) is applied to these relations to obtain the final set of authorizations. Each of these phases uses a distinct set of rules:

- Custom fact rules: This set of rules is implementation specific. Rules are written according to the needs of a particular system and affect the resulting authorizations by modifying, inserting or deleting facts in the session.

- Authorization rules: This set of rules computes $\mathcal{PAR}$ and $\mathcal{BAR}$ from $\mathcal{PCA}$, $\mathcal{ARCA}$ and $\mathcal{BARCA}$ that resulted from applying custom fact rules. In other words, this set of rules applies the logic defined in the rewriting rule (3.2). Since they are not implementation dependent, they could be easily replaced by regular code. The decision of using Drools to implement this set was made due to the simplicity of adding such rules to an existing session.

The resulting authorization set is computed from the results of processing custom fact rules, as illustrated in Figure 4.1. In practice, this means that authorization rules must be applied after custom fact rules. Otherwise, the process might return wrong results, for instance if a custom rule deletes a fact already used to build a $\mathcal{PAR}$ or $\mathcal{BAR}$ entry.

The desired order of execution is provided by the rule attribute *salience*. Authorization rules have a negative salience value, which means that they are run with lower priority than custom fact rules.

The next section illustrates, through some examples, how the custom fact rules are written. Section 4.2.2 describes the set of authorization rules.

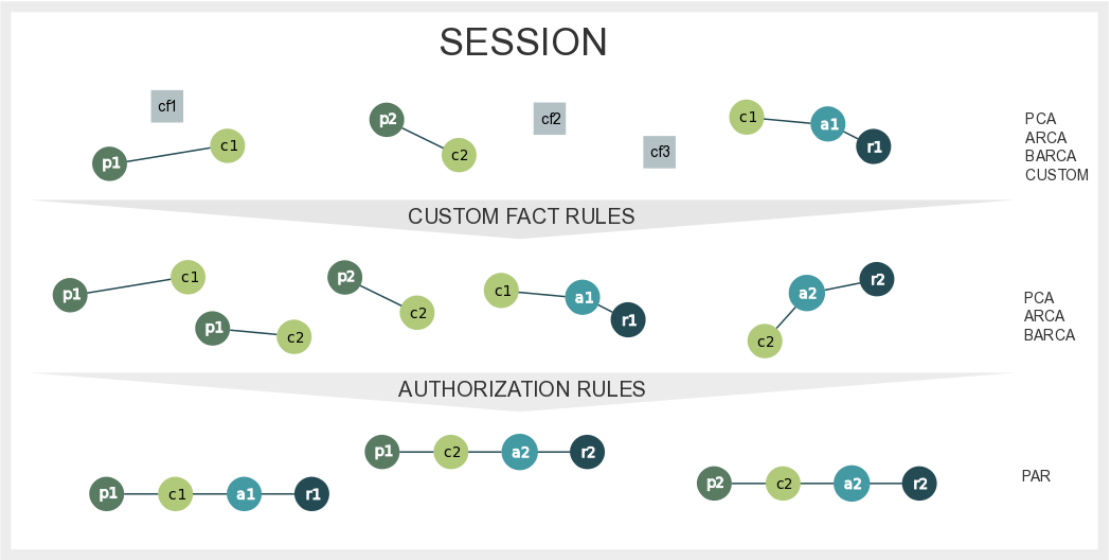

Figure 4.1: Drools Session: Rules Processing Sequence

4.2.1 Custom Facts

Custom facts are used to simulate system conditions that affect user authorizations. Their impact in the final set of authorizations depends on the logic described by the rules, as illustrated by the following examples.

Example 1. This scenario corresponds to the use case *Critical State* described in Section 3.3. It states that access to patient records, normally only available to the patient's responsible physician and nurse, should become available to the whole clinical staff if the patient is in critical condition. A simple way to achieve this is to map all clinicians (nurses or physicians) to a special category that has reading permissions on the record, if critical status is set. In a scenario where each session refers to a particular patient, and using the rewriting syntax described in [7], this logic can be expressed by the following rewriting rule:

$$\begin{aligned} pca(p) \rightarrow \quad & if\ isClinician(p)\ and\ isCritical() \\ & then\ append(categ(p), read\_all()) \end{aligned} \tag{4.1}$$

The same logic can be translated to the following rule:

```
rule "Rule critical state - read all"
  when
    CriticalState( criticalState == Boolean.TRUE )
    $principal : Principal()
    Category( $cid : id, id == "clinician" )
    $pca : Pca(
      principal.id == $principal.id,
      categories.containsOrEquals($cid, category.id)
    )
  then
    insert(
      new Pca(
        $principal,
        categories.getCategoryById("read_all")
      ) );
end
```

If there exists an instance of fact `CriticalState`, having property `criticalState` set to `Boolean.TRUE`, then every principal, associated to a specialization of `clinician`, is mapped to the category `read_all` (i.e. for each principal a `Pca` with category `read_all` is inserted).

Example 2. This scenario corresponds to the use case *Seal and Seal and Lock*, related to the ability of patients to restrict access to their clinical records. If a patient requests to *Seal and Lock* a part of his record then nobody can access it, no matter the circumstances. That result can be achieved by a rule that removes all mappings category/permission for locked resources:

```
rule "Sealed and Locked resources"
  when
    SealedResource ($resource: resource, locked == Boolean.TRUE)
    $arca : Arca(permission.resource.id == $resource.id)
  then
    delete($arca);
end
```

On the other hand, if a patient asks to *Seal* his record then data will be hidden by default, but clinicians can break that seal (in which case that action will be logged and the security manager notified). That process is also called Break The Glass (BTG) [31, 32, 33], as described in Section 3.3. The rule to implement that logic is more complex than the previous one because it has to make sure that only the principal who

performed the BTG gets access to the record. The following rule describes one possible way of doing it:

```
rule "Sealed resources"
  when
    $catSealed : Category(id == "sealed_resource")
    SealedResource (
      $resource : resource,
      locked == Boolean.FALSE
    )
    $arca : Arca(
      category.id != $catSealed.id,
      permission.resource.id == $resource.id
    )
    $pca : Pca(
      $principal : principal,
      category == $arca.category
    )
    BreakTheGlass(principal.id == $principal.id)
  then
    Permission permission =
      (Permission) PermissionFactory.buildPermission(
        $arca.permission.getAction(),
        $resource
      )
    delete($arca)
    insert(new Arca($catSealed, permission))
    insert(new Pca($principal, $catSealed))
end
```

This rules uses an auxiliary category `sealed_resources`. All `Arca` on sealed resources are removed from its original categories and mapped to this category. Then, every `Principal` that performed *Break the Glass* and was mapped to the original category is also mapped to `sealed_resources`.

Note that, even though all these rules apply very different logics, their consequences always create, modify or update `Pca`, `Arca` and `Barca`, i.e. these rules affect the mapping between principals and categories and between categories and permissions. The result of processing these rules is a set of facts that reflects the base relations and all the interactions introduced by the custom facts. That set is then processed by the rules that compute the resulting $\mathcal{PAR}$ and $\mathcal{BAR}$, as described in the next section.

4.2.2 Authorization

Authorization rules are used to compute $\mathcal{PAR}$ and $\mathcal{BAR}$ from the base facts in the session. In other words, this set of rules applies the logic defined in the rewrite rule (3.2). Since they are not implementation dependent, they could be easily replaced by regular code.

Authorizations are computed, from the results of applying custom fact rules, by the following pair of rules:

```
rule "Auth PAR"
  salience -100
  when
    $principal : Principal( $pid : id )
    $category : Category( $cid : id )
    $pca : Pca(principal.id == $pid, category.id ==  $cid)
    $arca : Arca(categories.containsOrEquals(category.id, $cid))
  then
    pars.add(
      new Par(
        $principal,
        categories.getPermissionChain($cid, $arca.category.id),
        $arca.permission
      ) )
end

rule "Auth BAR"
  salience -100
  when
    $principal : Principal( $pid : id )
    $category : Category( $cid : id )
    $pca : Pca(principal.id == $pid, category.id ==  $cid)
    $barca : Barca(categories.containsOrEquals($cid, category.id))
  then
    pars.add(
      new Par(
        $principal,
        categories.getProhibitionChain($cid, $barca.category.id),
        $barca.permission
      ) )
end
```

The first rule computes the authorizations and the second computes the prohibitions. Their conditions ensure that all `Pca` and `Arca` (`Barca` for prohibitions) with matching categories are found (like a database join). Their consequences insert a new `Par` in `Pars` for each match (`Pars` is the provided instance of `Pars()` that allows retrieving the

results from the engine).

Function `categories.containsOrEquals(a,b)` returns true if `a` and `b` are the same or if `a` is a superset of `b`. This function provides the hierarchical logic applied to categories. Parent categories are more general than their children (defining an *"is a"* relation between child and parent). This means that if a parent has an authorization then its children inherits it. Prohibitions, on the other hand, are propagated from more specific to more general categories, therefore the same function in rule `Prohibitions`, has the parameters switched (semantically equivalent to *"is contained or equals"*).

The pair of functions `getPermissionChain(a,b)` and `getProhibitionChain(a,b)` assures that `Pars` include the hierarchy for authorizations that are propagated. For instance, a *Par* in the form *{P, [C1, C2], Perm}* means that principal *P* is mapped to category *C1*, which has permission *Perm* but that permission was inherited from *C2*.

The two rules above implement almost directly the operational realisation of Axioms *(a1)*, *(a2)*, *(a3)* and *(a4)* given by the rewriting rule (3.2).

There is, however, a subtle but important difference between the rewrite specification and the rules above. The `if` clause in the rewrite specification implies that there are no conflicting authorizations in the result, i.e. an access request can not be authorized and prohibited, simultaneously.

This type of conflicts can be solved by either removing the conflicting authorization or the conflicting prohibition. The preferred solution depends on the implementation. We opted by adding two rules, `Auth conflict - Remove Arca` and `Auth conflict - Remove Barca`, that can be prioritized by configuration.

```
rule "Auth conflict - Remove Arca"
    salience configs.getSalienceConflictRemoveArca()
    when
        Category($cId : id)
        Action($aId : id)
        Resource($rId : id)
        $arca : Arca(categories.containsOrEquals(category.id, $cId),
          permission.action.id == $aId, permission.resource.id == $rId)
        Barca(category.id == $cId, permission.action.id == $aId,
         permission.resource.id == $rId)
    then
        delete($arca)
end
```

```
rule "Auth conflict - Remove Barca"
    salience configs.getSalienceConflictRemoveBarca()
    when
        Category($cId : id)
        Action($aId : id)
        Resource($rId : id)
        $barca : Barca(categories.containsOrEquals($cId, category.id),
          permission.action.id == $aId, permission.resource.id == $rId)
        Arca(category.id == $cId, permission.action.id == $aId,
         permission.resource.id == $rId)
    then
        delete($barca)
end
```

The `salience` value is configurable (as described in Section 6.3): value `arca` prioritizes authorizations over prohibitions and `barca` has the opposite effect.

If this configuration is set to `barca`, `getSalienceConflictRemoveArca()` returns a value bigger than `getSalienceConflictRemoveBarca()`, which means that rule `Auth conflict - Remove Arca` will run before `Auth conflict - Remove Barca` and conflicts will be resolved by removing authorizations. If the configuration is set to `arca` the result will be the exact opposite.

The conflict rules above, must be executed after custom fact rules and before pars rules. Therefore, `getSalienceConflictRemoveArca` and `getSalienceConflictRemoveBarca` must return negative values (but higher than the salience of `Auth PAR` and `Auth BAR`). The rules processing sequence can be now illustrated in more detail by Figure 4.2.

The conflict rules above can also be seen as the implementation of the operation ($\mathcal{OP}$), described in the distributed CBAC model and discussed in Sections 3.1 and 3.2, which is responsible for providing a single result from the composition of the results from different sites.

In fact, rules derived from different use cases should be treated as different sites even if they do not correspond to actual physical sites. Requirements for use cases, as the examples above, can have different “owners”. And specific entities may differ in different cases.

In the distributed approach, each site (or set of rules) is responsible for answering to access requests on its actions and permissions according to its requirements. The composition of answers from the different sites is delegated to a central point. This layered approach simplifies the task of rule management.

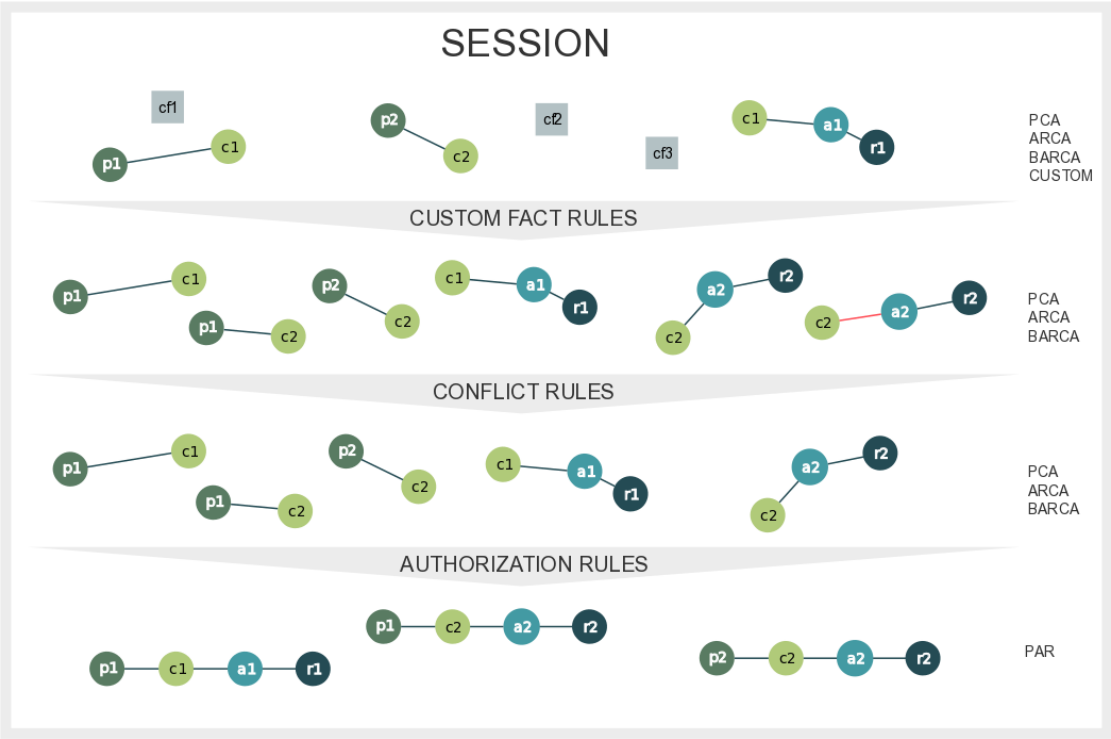

Figure 4.2: Drools Session: Complete Rules Processing Sequence

The implementation of the distribution version implies being able to define which custom rules are used at each site and which entities are mapped to each site. Even though the distributed version is not yet implemented in this work, the separation of the composition logic provided by the conflict rules is a good starting point for that evolution.

Chapter 5

The G-ACM Tool

The G-ACM was developed to fulfill the main goal of this work, which consists of the development of a tool to visualize CBAC policies as a graph, following [8].

Besides the graphical tool, or G-ACM Console, it also includes an engine to compute dynamic policy graphs as defined in [9], which we call G-ACM *Server*. Used together, these two components not only provide a way of visualizing static policy graphs but also provide some clues about how a policy managing tool can be used in systems that use dynamic policies.

The G-ACM prototype is accessible online at the following address:

`http://acm-joaosa.rhcloud.com/app/#/main`

The following Subsection describes the system from the conceptual point of view. Section 5.2 gives a brief explanation of the console's usage.

5.1 Conceptual model

As mentioned, the G-ACM Server implements the rewrite based semantics of the CBAC model. It uses the Drools engine to compute authorizations ($\mathcal{PAR}$ and $\mathcal{BAR}$) and relations $\mathcal{PCA}, \mathcal{ARCA}$ and $\mathcal{BARCA}$ from the base entities and a set of custom facts (and an optional set of base relations $\mathcal{PCA}$, $\mathcal{ARCA}$ and $\mathcal{BARCA}$). It exposes its logic through a set of services that allow client applications to:

- Get the list of principals, categories, actions and resources configured in the system.
- Get the list of configured custom facts and the possible values for their parameters.
- Compute $\mathcal{PCA}$, $\mathcal{ARCA}$, $\mathcal{BARCA}$, $\mathcal{PAR}$ and $\mathcal{BAR}$ from a set of facts applying the configured rules described in Chapter 4.

The G-ACM Console provides a graphical representation of relations $\mathcal{PCA}$, $\mathcal{ARCA}$, $\mathcal{BARCA}$, $\mathcal{PAR}$ and $\mathcal{BAR}$, showing authorizations for a specific system state. Additionally, it uses the above services to expose available custom facts that correspond to the system parameters that affect authorizations. The user can simulate authorizations for different system states by choosing the input facts for the authorization service.

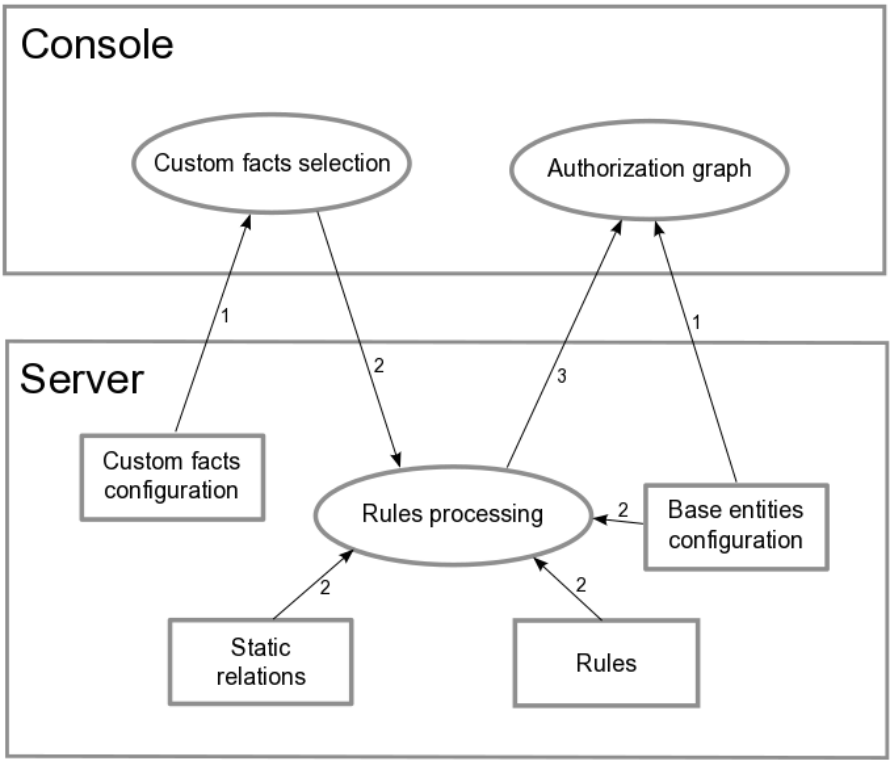

Figure 5.1: G-ACM: Conceptual Model

Figure 5.1 shows the two components from a conceptual perspective. The numbers in the diagram represent the sequence of events for a typical interaction between the two components:

1. The console uses the services to get the base entities and the custom facts configured in the system.

2. The authorization service is called, generating a Drools session that is fed with the custom facts chosen by the user, the base entities, the base relations and the configured rules.

3. The console displays the resulting authorizations and relations.

5.2 Console Usage

This section describes how authorizations are represented in the G-ACM Console and the main features available.

5.2.1 Authorization Graph

The G-ACM Console explores the representation of access control policies, presented in [8], where authorizations are represented as a graph having nodes of types *P, C, A* and *R* corresponding to entities *principal, category, action* and *resource*, respectively. A path of length three with edges *PC, CA* and *AR* represents an authorization, i.e. the set of paths in this form represents the relation $\mathcal{PAR}$.

Note that, not all entities are part of an authorization or prohibition. For instance, a principal may not be assigned to any category or a category may not be linked to any principal or permission. The same is true for relations $\mathcal{PCA}$, $\mathcal{ARCA}$ and $\mathcal{BARCA}$. Even though $\mathcal{PAR}$ and $\mathcal{BAR}$ may be inferred from these relations, that does not mean that all elements in these relations are part of an authorization or prohibition (e.g., a principal may be linked to a category that has no permissions or a category may have permissions but no principals assigned).

Furthermore, due to the dynamic behaviour of the model, elements not included in permissions or prohibitions may change with the changes in context.

From the authorization management perspective, it is useful to have a complete view of entities and relations. Therefore the graph displays all entities and relations $\mathcal{PCA}$, $\mathcal{ARCA}$ and $\mathcal{BARCA}$, as shown in Figure 5.2, using colours to tell apart nodes and

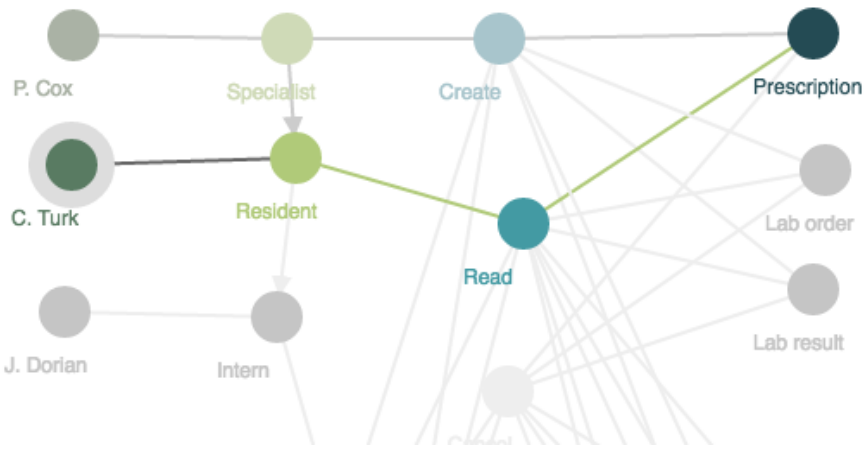

Figure 5.2: Detail of an authorization graph

links in authorizations from the rest. Colours of nodes and links in the graph have the following meanings:

- The light grey nodes and edges represent the set of base entities and relations that are not part of any authorization.
- Nodes included in any authorization are displayed in colours, the edges in authorization graphs are represented in a darker shade of grey.
- The nodes in the selected path are shown in brighter colours (in this example, the path [P.Cox, Specialist, Resident, Create, Lab Order]).
- Edges in the selected path are represented in a even darker shade of grey. Except for edges of type *CA* and *AR* that have specific rules:
  - Authorizations are represented in green.
  - Prohibitions are represented in red.
  - If the link is representing simultaneously an authorization (or more) and a prohibition (or more) than it is represented in grey such as all other edge types.

According to these rules, authorizations are represented as shown in Figure 5.3:

The $\subseteq$ relation between categories can be represented as a directed path between two nodes *CC*. Therefore valid authorizations can have any number of *CC* edges, as

Figure 5.3: Permission

shown in Figure 5.4. This representation provides information on authorizations but also on their propagation through the $\subseteq$ relation between categories. In this case, the permission is transmitted to principal P. Cox through the chain $Specialist \subseteq Resident \subseteq Intern$. Note that, *Specialist* is a specialization of *Resident*, which specializes *Intern*. More broadly, this relation is defined semantically as an *is a* relation between categories (in this case a $\subseteq$ relation), as required by Axiom *(a1)*.

Figure 5.4: Inherited Permission

Prohibitions (relation $\mathcal{BAR}$) are represented in a similar way, but the links *CA* and *AR* are displayed in a different colour as depicted in Figure 5.5.

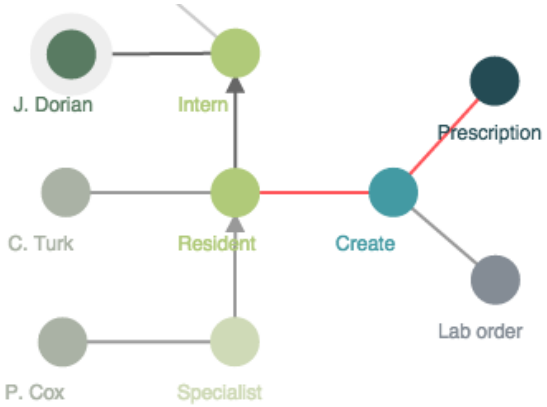

Figure 5.5: Inherited Prohibition

This graph results from the following pair of authorizations:

```
[ C.Turk, Resident, Create, Lab Order ] : Authorization
[ C.Turk, Resident, Create, Prescription ] : Prohibition
```

This, following the rules of inheritance described in 4.2.2, generates the additional pair:

```
[ P.Cox, [Specialist, Resident], Create, Lab Order ] : Authorization
[ J.Dorian, [Intern, Resident], Create, Prescription ] : Prohibition
```

Note that prohibitions are propagated in the inverse direction of authorizations. In this case the prohibition is propagated from the *Specialist* to the *Intern*, if the former is not allowed to create prescriptions then neither is the latter.

The selected path is the authorization subgraph that contains the node selected by the user (the node with a large grey circle around it). Figure 5.6 shows the same example as in Figure 5.5 with the difference that the selected node is now the action *Create*.

All nodes and links are now selected because the selected node, action *Create*, is part of all authorizations. Furthermore, link *(Resident, Create)* is shown in grey because it is representing both a prohibition and a authorization.

5.2.2 Custom Fact Selection

As mentioned, in the dynamic CBAC, the assignment of principals and permissions to categories can depend on the system's state. This characteristic provides great

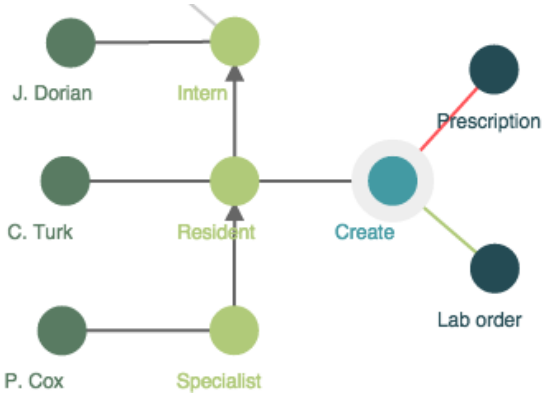

Figure 5.6: Authorization Graph: Selection on Action

flexibility, but introduces some degree of complexity when managing and verifying authorizations. The effective set of authorizations depends on the system's state at that moment and on the logic associated to each of the parameters used by the authorization engine. Different system parameters can have cross effects making it difficult to predict which will be the authorizations for a specific set of parameter values.

The G-ACM console helps dealing with that complexity by providing a form of simulating scenarios for the different parameters involved in the authorization evaluation. The left pane, accessible through the button in the top left corner of the screen, gives access to the list of available custom facts. Figure 5.7 shows the steps needed to select a fact and the values of its parameters (for details on the configuration of custom facts please consult Section 6.3). After selection, the button UPDATE submits the chosen facts and parameter values to the engine and updates the graph with the new set of authorizations.

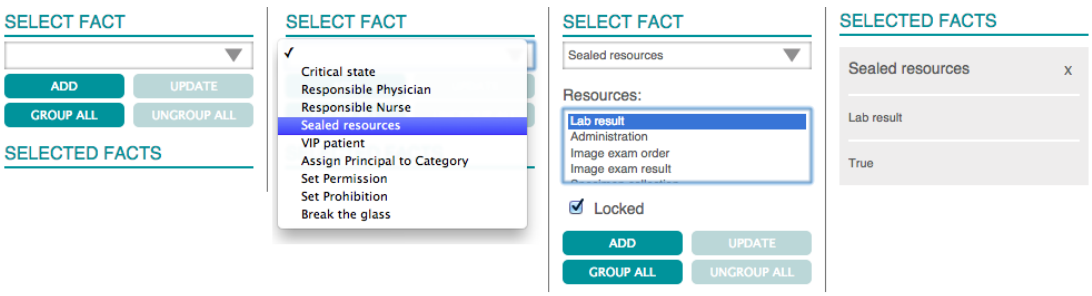

Figure 5.7: Custom Fact Selection Sequence

5.2.3 History

This feature was introduced to help the user to analyze the effects of context changes on authorizations.

When a graph is updated the data from the previous graph is stored in a list. This list is available in menu HISTORY in the pane on the right side of the screen, as shown in Figure 5.8.

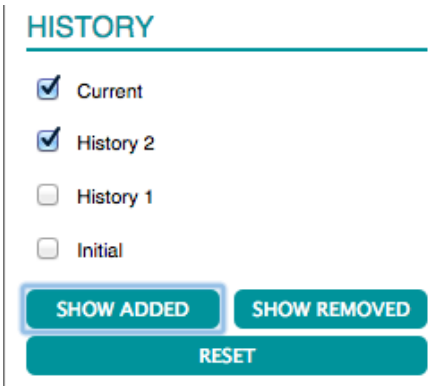

Figure 5.8: Graphs History Menu

If the user selects two entries/graphs from the list, then the buttons below become active. SHOW ADDED displays the authorizations and relations that exist in the most recent graph and did not exist in the older one. SHOW REMOVED does the opposite: the resulting graph displays the set of authorizations and relations that existed in the older one and do not exist in the most recent.

A double click on an item in the list will show the corresponding graph (the list of selected facts is also updated to the set that was used to generate that graph). Pressing RESET is equivalent to double click on the Current item in the list, restoring the most recent graph.

5.2.4 Nodes Grouping

Nodes Grouping improves graphs readability, in particular in large graphs, when there are many entities with the same set of links.

Button GROUP ALL groups all nodes that have the same type and the same exact set

of neighbours (have links to the same set of nodes).

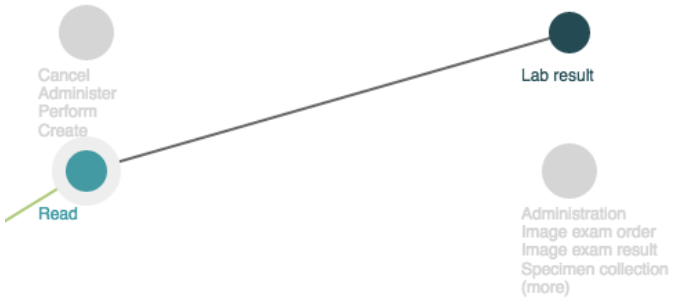

Figure 5.9: Grouped Nodes

The set of nodes in the same group (or cluster) is represented by a larger circle than regular nodes. Figure 5.9 shows the detail of a graph with two regular nodes and two clusters. The cluster label lists the name of its elements. For clusters with more than four elements the list can be made visible by double clicking on the node.

Button UNGROUP ALL replaces the cluster nodes by their elements, restoring the original form of the graph .

5.2.5 Settings

The right pane includes some additional settings:

- *Type of Layout*: this option allows changing between a regular graph layout and a D3 force layout. In the normal layout nodes are positioned according to their types and hierarchy (for categories). In the force layout nodes are positioned by gravitational forces. This is part of the study on variations of the form of displaying the graph and is still work in progress.
- *Show Labels*: this check box shows/hides the labels next to nodes.
- *Conflict priority*: this configuration allows changing the conflict resolution strategy, described in Section 4.2.2. Option ARCA means that in case of a conflict, authorizations have priority over prohibitions and BARCA has the opposite effect. This is a server side configuration, it becomes active only after pressing the UPDATE button and will only affect next updates to the graph.

- *Menu* FILTER: This menu (Figure 5.10) allows hiding/showing the relation in the graph by type. This is useful to improve the graph readability when the number of nodes and links increases.

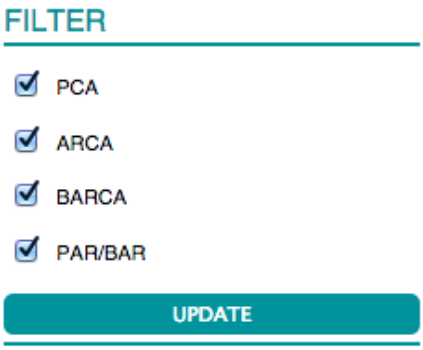

Figure 5.10: Relations Filter

Chapter 6

Technical Implementation

This section describes in detail the technical implementation of the G-ACM. First, we portrait the system's architecture, then we describe the tools used in its implementation, and finally we provide a detailed description of the technical aspects of each component.

6.1 Architecture

The G-ACM has two main components: the server that computes the dynamic policy graphs, and the console that provides the user interface of the system and displays the policies in a graphical format.

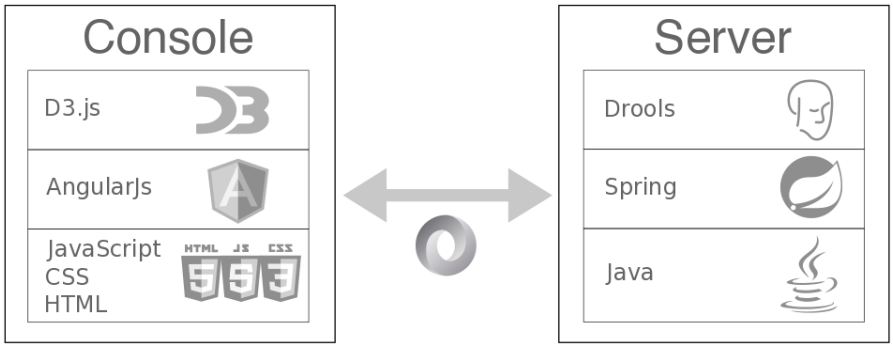

Figure 6.1: High Level Architecture

The console uses web technologies so that it can run in a web browser. The engine

was developed in Java and uses the Drools rule engine to compute the authorizations. Figure 6.1 resumes the technologies used in each layer.

The console was developed in HyperText Markup Language (HTML) and JavaScript and uses two main frameworks: *Angular JS* [37] and *D3.js* [38]. *Angular JS* is a Model-view-controller (MVC) and an Inversion of control (IoC) platform and *D3.js* is used to draw the main graph.

The server side, developed in Java, uses Spring [39] for IoC and for exposing services and uses the Drools rule engine to compute authorizations.

6.2 Chosen Tools

Spring The Spring Framework is an open source application framework and IoC container for Java. Spring was developed as replacement of Enterprise JavaBeans (EJB) and became popular in the Java community.

Besides the IoC container, which is central to the framework, Spring offers many different features, such as: aspect-oriented programming, data access, transaction management, MVC framework, among others.

The G-ACM uses the IoC container for Dependency Injection (DI) and the MVC framework to create the services that are used to communicate with the server.

- *IoC container*: the IoC container is central to the framework, it provides a mechanism for configuring and managing Java objects. Applications that use Spring rely on the container for managing object lifecycles: configuring objects, creating them, calling their initialization methods, and wiring them together.

  Objects configured by the framework are also called managed objects or *beans*. Beans definitions are provided to the container either by XML or code annotations.

  Beans can be accessed through dependency lookup or DI. Dependency lookup allows requesting objects from the container by name or type. DI, on the other hand, is a pattern where the container passes objects by name to other objects, via either constructors, properties, or factory methods.

  The G-ACM Server uses extensively Spring's DI mechanism that largely reduces the coupling between components and simplifies their replacement.

- *Spring's Web MVC*: the Spring's Web MVC [40] component is used to expose the Hypertext Transfer Protocol (HTTP) services used to communicate with the server. This framework is request-driven, a central *Servlet* receives requests and dispatches them to controllers.

  Controllers are defined through an annotation-based programming model and do not have to extend specific base classes or implement specific interfaces.

  Section 6.3 describes, in detail, how this component is used.

AngularJS AngularJS (or Angular) is a framework to build dynamic web applications. It uses HTML as a template language and extends HTML's syntax to express application's components. It provides a way of building applications in a well-defined structure and avoids having to write all the Document Object Model (DOM) and AJAX code by hand. The main features include: data-binding, basic templating directives, form validation, routing, deep-linking, reusable components and DI.

The level of abstraction of Angular makes it a good fit for CRUD applications. The authorization graph in the G-ACM requires intensive DOM manipulation, which is not something at which Angular excels. For that reason a specific library (see paragraph **D3**, below) was used to build the authorization graph and Angular is used in the rest of the application.

D3 D3 [38] is a JavaScript framework that allows binding data to a DOM and apply data driven transformation to the document.

The G-ACM uses D3 to draw the authorization graph.

6.3 Server

The G-ACM's server component is a web application developed in Java. Maven [41] is used for build management. The project is divided in four modules organized as depicted in the following diagram:

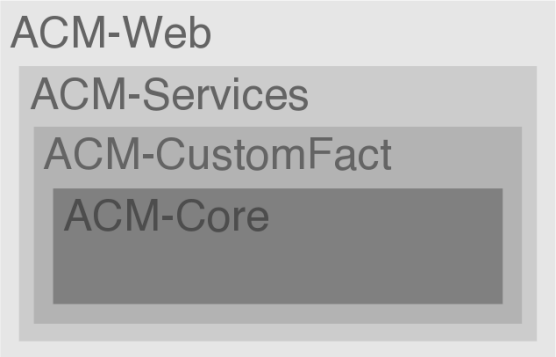

Figure 6.2: G-ACM Modules

This organization derives from the role that each module plays in the application:

- *ACM-Core* loads the rules and base entities into memory and provides the logic responsible for the computation of authorizations.
- *ACM-CustomFacts* allows configuring the available parameters that affect rules evaluation.
- *ACM-Services* provides the services that allow interaction with the application.
- *ACM-Web* contains the web application configuration, namely the *web.xml* deployment descriptor that initializes Spring's listener and dispatcher Servlet.

To compile the project using Maven, one should run the following command from the project's root directory:

```
mvn clean install -U
```

The resulting war (web application archive) file is written to the folder `/ACM-Web/target`.

As mentioned before, the server component of the G-ACM uses Drools to implement the rewrite based semantics of the CBAC model and provides services that expose

the system's configuration and allow requesting the computation of authorizations for a specific set of parameters. It can be divided in the following set of high level features:

- *Base configuration*: loads into memory the base configuration (principals, categories, actions and resources).
- *Custom facts configuration*: configures the implementation specific parameters that affect rule evaluation (i.e., configure the possible set of parameters that define the rule execution context).
- *Rule processing*: computes $\mathcal{PAR}$, $\mathcal{BAR}$, $\mathcal{ARCA}$, $\mathcal{BARCA}$ and $\mathcal{PCA}$ according to implementation specific rules, the provided context and Axioms *(a1)-(a4)*.
- *Services API*: provides a set of services that expose the base configuration, the custom facts configuration and the services to compute authorizations.

In the remaining of this section we describe in detail the implementation of these features. Since they all rely on Spring's DI mechanism, we start by describing how that mechanism is used.

Spring container and Spring beans Spring container is initialized by the inclusion of the listener `ContextLoaderListener` in the applications web.xml file.

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>classpath:spring/application-config.xml</param-value>
</context-param>

<listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>
```

Parameter `contextConfigLocation` provides the location of the Spring's configuration file, named `application-config.xml` in this case. That file includes the following entry:

```
<context:property-placeholder
  location="classpath:acm-application.properties, classpath:acm-facts.properties" />
```

The `context:property-placeholder` entry provides a way of loading key-value property files to the container, which can be later used in other configurations. The rest of the

lines of the `application-config.xml` are responsible for importing other configuration files, where the actual *Spring beans* are configured.

Spring beans, or *beans*, are the Java objects that are managed by the Spring container. These are normal Java classes that are initialized through the Spring configuration and that can be used by Spring's DI mechanism.

Beans can be declared by annotation or in XML files. In the G-ACM, the configuration through XML was used.

The following example shows the configuration file used in the G-ACM to define the beans related to custom facts. The beans that hold the lists of principals and categories are defined in the first two `bean` blocks. Since the constructors of the corresponding classes need the name of the file containing the data, both have a `constructor-arg` attribute that is used to inject the name of the configuration files that hold the lists of principals and categories.

```
...
<bean id="principals" class="acm.core.fact.Principals">
  <constructor-arg type="java.lang.String" value="${file.principal}"/>
</bean>
<bean id="categories" class="acm.core.fact.Categories">
  <constructor-arg type="java.lang.String" value="${file.category}"/>
</bean>
...
<bean id="pcas" class="acm.core.fact.Pcas">
  <constructor-arg type="java.lang.String" value="${file.pca}"/>
  <constructor-arg ref="principals"/>
  <constructor-arg ref="categories"/>
</bean>
...
```

This configuration makes sure that, at the application startup, beans `principals` and `categories` are created. In Spring, beans are singletons by default, so this will also make sure that only a single instance of each class will be available in the system.

The third `bean` block describes how the class `Pcas`, which holds the initial mapping between principal and categories should be instantiated. In this case, the constructor of class `Pcas` has three parameters: besides the file name, as in `principals` and `categories`, it has one parameter of type `Principals` (`acm.core.fact.Principals`) and another of type `Categories`. Therefore instead of providing values, the configuration provides references to beans `principals` and `categories` defined previously.

The values of the parameter of type `String` are in the form `${key}`, this means that the Spring container will replace those keys by the corresponding values loaded by the `context:property-placeholder` configuration described above.

Base configuration On startup the core module loads the base entities (`Principal`, `Category`, `Action` and `Resource`). These entities are described in a set of *JSON* files. For instance, the list of principals is defined in the file *principal.json*, which contains a list of objects of the form:

```
{   "id": "000001",
    "name": "P. Cox",
    "title": "MD"    }
```

These lists are loaded to singletons `Principals`, `Categories`, `Actions` and `Resources`, which are initialized through the following Spring configuration (file `fact-config.xml`):

```
<bean id="principals" class="acm.core.fact.Principals">
    <constructor-arg type="java.lang.String" value="${file.principal}"/>
</bean>
<bean id="categories" class="acm.core.fact.Categories">
    <constructor-arg type="java.lang.String" value="${file.category}"/>
</bean>
<bean id="actions" class="acm.core.fact.Actions">
    <constructor-arg type="java.lang.String" value="${file.action}"/>
</bean>
<bean id="resources" class="acm.core.fact.Resources">
    <constructor-arg type="java.lang.String" value="${file.resource}"/>
</bean>
```

Note that the constructor of each of these classes accepts the file name for the corresponding *JSON* configuration file.

Besides the initial entities, the system allows configuring initial sets for the relations principals/categories, categories/permissions and categories/prohibitions. These configurations define a base set of $\mathcal{PCA}, \mathcal{ARCA}$ and $\mathcal{BARCA}$ as in a RBAC system (more specifically an Hirearchical RBAC since categories can be described in a hierarchical fashion).

The file for $\mathcal{PCA}$ contains a list of pairs principal/category. For example, the fact that principal with id `000001` is mapped to the category `physician_specialist`, is represented by:

```
{   "principal": "000001",
    "category": "physician_specialist"   }
```

Mapping category/permission and category/prohibition have similar layouts. For example, the following configuration defined in the file for $\mathcal{ARCA}$ expresses the members of the category `clinician` are authorized to read prescriptions. The same configuration in the file for $\mathcal{BARCA}$ means that the members of category `clinician` are prohibited to read prescriptions.

```
{   "category": "clinician",
    "action": "read",
    "resource": "prescription"  }
```

These configurations are loaded from a set of *JSON* files to the singletons `Pcas`, `Arcas` and `Barcas` through a Spring configuration in file `fact-config.xml`:

```
<bean id="pcas" class="acm.core.fact.Pcas">
    <constructor-arg type="java.lang.String" value="${file.pca}"/>
    <constructor-arg ref="principals"/>
    <constructor-arg ref="categories"/>
</bean>
<bean id="arcas" class="acm.core.fact.Arcas">
    <constructor-arg type="java.lang.String" value="${file.arca}"/>
    <constructor-arg ref="categories"/>
    <constructor-arg ref="actions"/>
    <constructor-arg ref="resources"/>
</bean>
<bean id="barcas" class="acm.core.fact.Barcas">
    <constructor-arg type="java.lang.String" value="${file.barca}"/>
    <constructor-arg ref="categories"/>
    <constructor-arg ref="actions"/>
    <constructor-arg ref="resources"/>
</bean>
```

Like base entities, the constructor of each class also receives the name of the configuration file. In this case, however, those constructors receive additional parameters: bean `principals` and bean `categories` are injected in `Pcas`; `Arcas` and `Barcas` receive beans `categories`, `actions` and `resources`.

It is important to note that the implementations of `Principals`, `Categories`, `Actions`, `Resources`, `Pcas`, `Arcas` and `Barcas` can be changed by updating the attribute `class` and constructor arguments in the configurations above. A plausible scenario for replacing

these implementations would be reading the base configurations from different data sources (e.g. a database or a RBAC system). As shown in Figure 6.3, the only restrictions on new implementations is to implement the interface `FactsListInterface`.

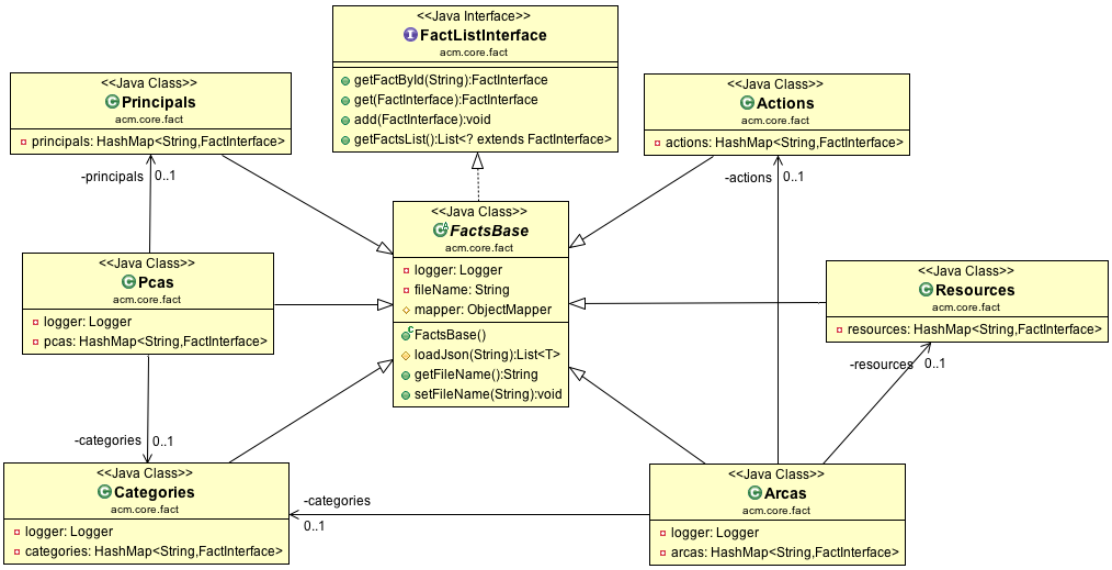

Figure 6.3: Classes Diagram: Base Configuration

Custom facts configuration In Drools, the data used in the evaluation of rules are instances of regular Java objects and are called *facts* (see Section 4.1 for further detail).

The G-ACM defines three types of facts: instances of the base entities (`Principal`, `Category`, `Action`, `Resource`), the base relations (`Pca`, `Arca` and `Barca`), and a set of configurable entities that we call *custom facts*. Upon user input, custom facts are added to the rules execution context affecting the resulting authorizations. This provides the dynamic behaviour in the G-ACM.

Unlike base entities and base relations, which are part of the definition of the CBAC model, custom facts are implementation specific, therefore available custom facts, their parameters, and their possible values must be configurable.

Property `file.custom.fact.config` specifies a *JSON* file that describes the custom facts available in the system. The following sample exemplifies the configuration for the fact `RESPONSIBLE_PHYSICIAN`:

```
{
    "id": "RESPONSIBLE_PHYSICIAN",
    "description": "Principal is the physician responsible for the patient",
    "label": "Responsible Physician",
    "single": false,
    "className": "acm.core.fact.custom.ResponsiblePhysician",
    "parameters": [
        {
          "type": "SELECTION",
          "index": 0,
          "label": "Responsible physician",
          "description": "Responsible physician",
          "valuesType": "PRINCIPAL"
         }
    ]
}
```

Property `id` identifies the custom fact type and has to be unique (if there are two configurations with the same id then the second one will overwrite the first). Property `label` is the human readable identifier for the custom fact type and `description` provides its meaning. Property `single` indicates if multiple instances or just a single one are acceptable in each session. Property `className` specifies the object that this fact corresponds to.

Array `parameters` defines the list of parameters of a fact (a single one, in this case). The type `SELECTION` indicates that the property accepts a value from a predefined set (typically displayed as a drop-down list). Property `index`, identifies the position in the list of parameters, the first parameter must have index 0, the second 1 and so on. Property `valuesType` indicates the type of values that the parameter accepts. These values are also implementation specific, therefore value types are also configurable. Property `file.custom.fact.parameterValues.config` specifies a *JSON* file to configure available value types, which are defined as entries of the form:

```
{
    "id": "PRINCIPAL",
    "bean": "principalValues"
}
```

Property `id` identifies the type of value, and `valuesType` in the custom fact configuration must refer to one of these ids. Property `bean` is the bean that implements the class as defined in Spring's configuration file `customFacts-config.xml`:

```
<bean id="principalValues" class="acm.core.fact.custom.config.param.values.PrincipalValues">
```

```
    <constructor-arg ref="principals"/>
</bean>
```

Parameter values beans are obtained at runtime by their name through the method `getParameterValuesBean` in the helper class `ParameterValuesBeanHelper`.

Figure 6.4 shows the classes diagram for base facts and for two examples of custom facts: `CriticalState` and `SealedResource`. It is possible to see that both types implement the interface `FactInterface`, which has a single method: `getId()`. That method is used during rule evaluation to identify fact instances, therefore it must return a unique identifier for each instance of a fact.

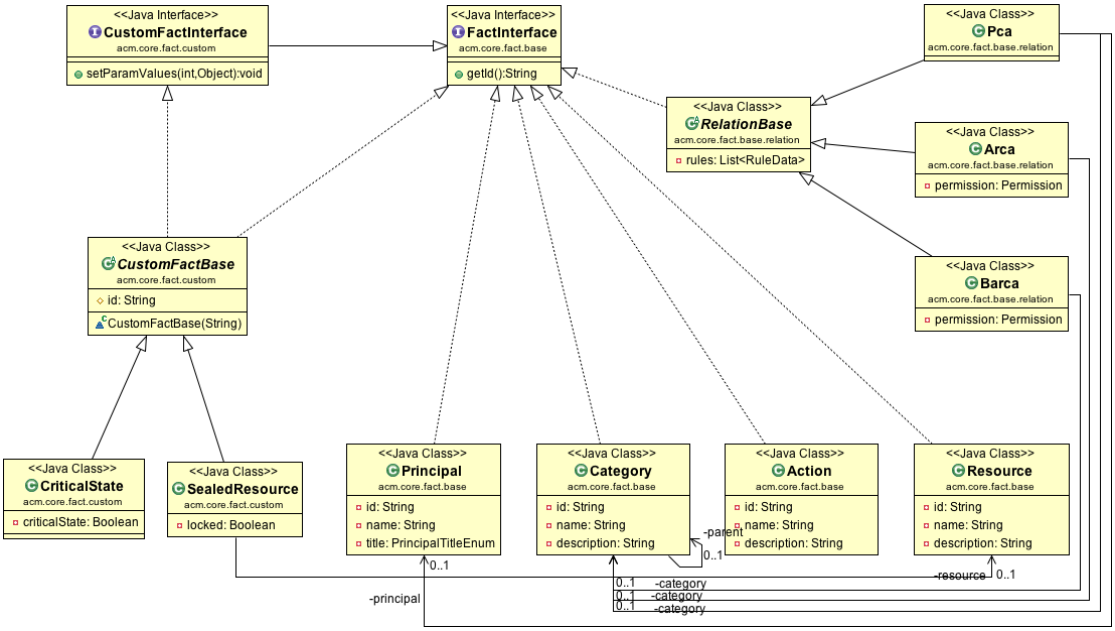

Figure 6.4: Classes Diagram: Fact Interface

Classes `CustomFactsConfig` and `ParameterValuesConfig` load the respective *JSON* files on startup. These classes are instantiated by the following Spring configuration, which also creates a singleton for `CustomFactFactory`:

```
<bean id="parameterValuesConfig"
   class="acm.core.fact.custom.config.param.values.ParameterValuesConfig">
  <constructor-arg type="java.lang.String" value="${file.custom.fact.parameterValues.config}"/>
</bean>
```

```
<bean id="customFactsConfig" class="acm.core.fact.custom.config.CustomFactsConfig">
  <constructor-arg type="java.lang.String" value="${file.custom.fact.config}"/>
  <property name="parameterValuesBean" ref="parameterValuesBeanHelper" />
  <property name="parameterValuesConfig" ref="parameterValuesConfig" />
</bean>

<bean id="customFactFactory" class="acm.core.fact.custom.CustomFactFactory">
  <property name="customFactsConfig" ref="customFactsConfig" />
  <property name="parameterValuesConfig" ref="parameterValuesConfig" />
</bean>
```

`CustomFactFactory`, as shown in Figure 6.5, uses these configurations to build new custom fact instances at runtime: from property `className` it instantiates the class using reflection and uses function `setParamValues` from `CustomFactInterface` (shown in Figure 6.4) to set the values for the parameters.

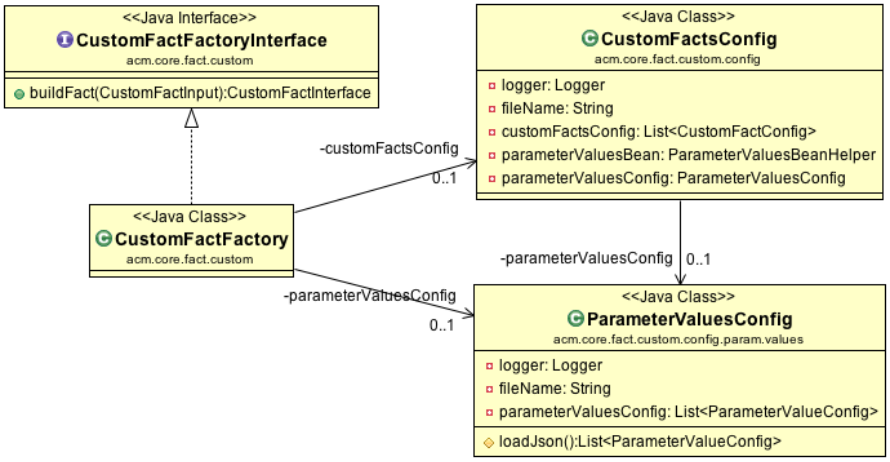

Figure 6.5: Classes Diagram: Custom Facts Configuration

The values for the custom fact parameters are chosen by the user from the set of acceptable values, which are provided by the method `getValues()` defined in interface `ParameterValuesInterface` (see Diagram 6.6). For example, class `PrincipalValues` returns the list of principals' ids configured in the system whereas `BooleanValues` return a list with values `true` and `false`.

The other method in the same interface, `getObjectByValue(String value)` is used by `CustomFactFactory` to get the object instance for the selected value. For instance, in class `PrincipalValues` it returns the instance of class `Principal` that has the provided id.

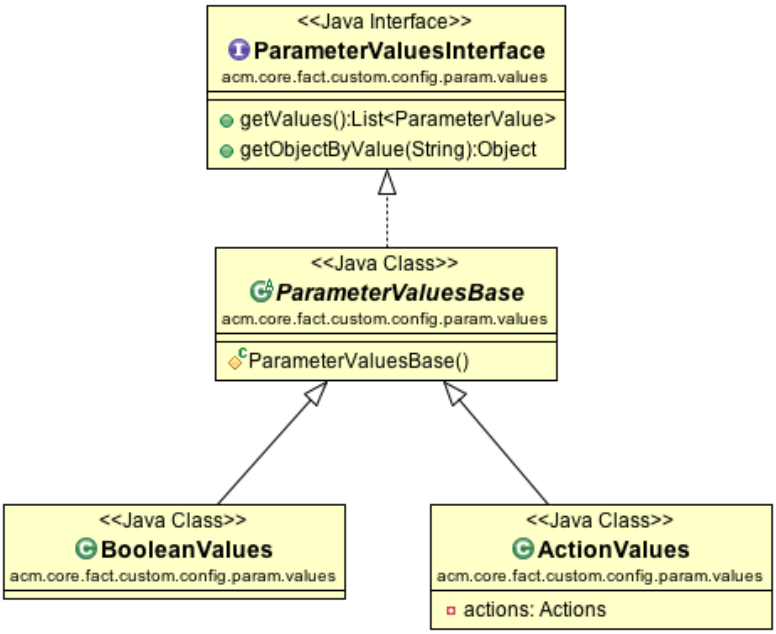

Figure 6.6: Classes Diagram: Custom Facts Parameters Configuration

It is important to note that a new custom fact, using available value types, can be added in two steps: create an implementation for `CustomFactInterface` and add a new entry to the custom fact configuration file.

New value types can be added in three additional steps: provide a new implementation for `ParameterValuesInterface`; add a new bean entry for that class in the Spring configuration file `customFacts-config.xml`; and add the new parameter value to the *JSON* file `file.custom.fact.parameterValues.config`.

Rule processing Class `ParsCore` is responsible for the orchestration of the rule processing sequence. It includes properties that store the list of rule files and hold references to the sets of base facts and relations. This class is instantiated, and its properties are set by the following Spring configuration:

```
<bean id="parsCore" class="acm.core.ParsCore">
    <property name="configs" ref="configs"/>
    <property name="principals" ref="principals"/>
    <property name="categories" ref="categories"/>
    <property name="actions" ref="actions"/>
    <property name="resources" ref="resources"/>
```

```
    <property name="pcas" ref="pcas"/>
    <property name="arcas" ref="arcas"/>
    <property name="barcas" ref="barcas"/>
     <property name="rulesFiles">
      <list  value-type="java.lang.String">
        <value>${file.rules.pars}</value>
        <value>${file.rules.custom}</value>
       </list>
    </property>
</bean>
```

Properties `file.rules.pars` and `file.rules.custom` specify the files that contain the rules described in Section 4.2. The first contains authorization rules and the second custom fact rules.

Method `runRules`, shown below, accepts a list of custom facts and triggers the process of rules evaluation.

```
public Pars runRules(List<FactInterface> customFacts) {

    ParsCoreThread parsCoreThread = new ParsCoreThread(configs, principals,
            categories, actions, resources, pcas, arcas, barcas,
            customFacts, rulesFiles);

    // Observer for results
    parsCoreThread.addObserver(this);

    Thread tParsCore = new Thread(parsCoreThread);
    tParsCore.run();

    return getResults();
}
```

This method, creates a thread `ParsCoreThread`, providing the list of facts, custom facts, and rule files and uses Java's pattern *observer/observable* to get the results from the thread when the processing is finished.

`ParsCoreThread` constructor builds a `ParsRuleSet`, that groups the data as required by the Drools session API:

- `List<FactInterface> facts`: a list that holds all facts to be inserted in the session.
- `String[] rules`: an array containing the names of all rule files.

- `HashMap<String, Object> globals`: this map includes all data needed to evaluate rules that are not facts (i.e. do not affect the computed results). In this case those are the base entities, configurations and an instance of `acm.core.fact.Pars` that is used to retrieve the results.

`ParsCoreThread run` method creates an instance of `StatefulRuleRunner` that creates a Drools stateful session to process the rules (for more information on Drools stateless and stateful sessions consult Drools documentation [35]).

Upon completion, the results are retrieved using `ParsRuleSet` method `getResults()`. Figure 6.7 shows the main classes used to process rules in the G-ACM.

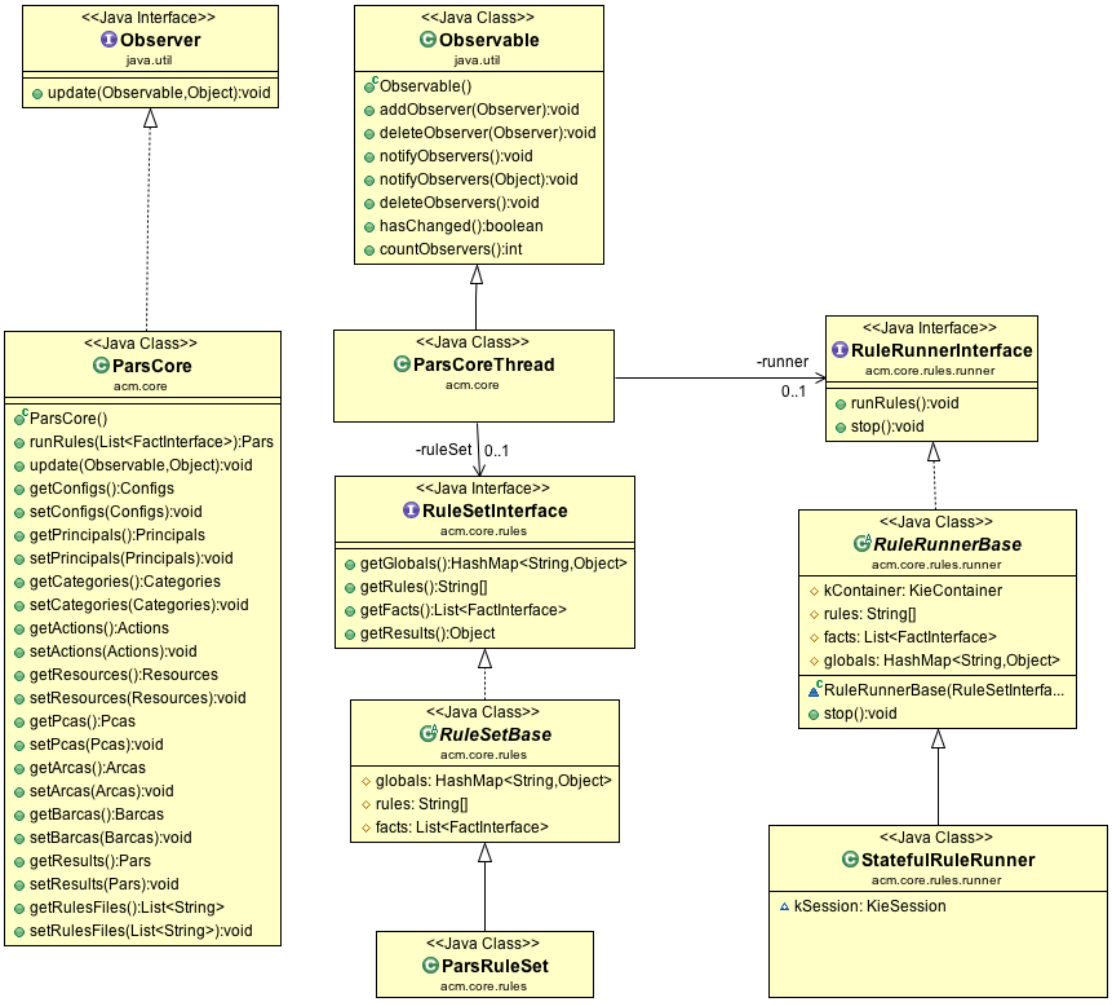

Figure 6.7: Classes Diagram: Rules Processing

Services API As mentioned in Section 6.2, G-ACM services are implemented on top of Spring MVC. This framework is request-driven: a central *Servlet* receives requests and dispatches them to controllers. The `DispatcherServlet` is a regular servelt (inherits from the `HttpServlet` base class) and, as such, is declared in the application's `web.xml`:

```
<servlet>
    <servlet-name>dispatcherServlet</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/mvc-config.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
    <servlet-name>dispatcherServlet</servlet-name>
    <url-pattern>/</url-pattern>
</servlet-mapping>
```

The `url-pattern` attribute is set to "/" which means that this servlet handles all incoming requests. Parameter `contextConfigLocation` specifies the location of the configuration file for the beans to be loaded at the servlet startup. Controllers are configured as any other beans (see paragraph **Spring container and Spring beans**, above). In the G-ACM this file contains the configuration for the controller classes that handle requests related to base entities, custom facts configuration, authorizations, and configurations, (`BaseModelController`, `CustomFactsController`), `AuthorizationController`, and `ConfigController`, respectively)

These controllers are instantiated by Spring through the following configuration in file `mvc-config.xml`:

```
<bean id="baseController" class="acm.services.base.BaseModelController">
    <property name="sites" ref="sites"/>
    <property name="principals" ref="principals"/>
    <property name="categories" ref="categories"/>
    <property name="actions" ref="actions"/>
    <property name="resources" ref="resources"/>
</bean>

<bean id="customFactController" class="acm.services.fact.CustomFactController">
    <property name="customFactsConfig" ref="customFactsConfig"/>
</bean>
```

```
    <bean id="authorizationController" class="acm.services.auth.AuthorizationController">
        <property name="customFactFactory" ref="customFactFactory"/>
        <property name="parsCore" ref="parsCore"/>
    </bean>

    <bean id="configsController" class="acm.services.config.ConfigController">
        <property name="configs" ref="configs"/>
    </bean>
```

Spring provides an annotation-based programming model to implement MVC controllers. Controllers implemented in this style do not have to extend specific base classes or implement specific interfaces.

Annotation `@Controller` is used to indicate that a class serves the role of a controller. The dispatcher scans classes with this annotation and detects the methods annotated with `@RequestMapping` and maps them to HTTP requests.

Consider, for instance, method `getParamOptions` in class `CustomFactController` that provides the possible values for the parameters configured for each custom fact:

```
...
@Controller
public class CustomFactController {

    @RequestMapping("/customFacts/{factId}/params/{rank}/options")
    public @ResponseBody OptionDTO getParamOptions(@PathVariable String factId,
            @PathVariable int rank) {
        CustomFactEnum customFact = CustomFactEnum.getEnumByName(factId);
        return new OptionDTO(factId, rank, customFactsConfig.getOptionsList(
                customFact, rank));
    }

    /* Other methods... */
}
...
```

Besides the annotation `@Controller` that defines this class as a controller, annotation `@RequestMapping` above ensures that the HTTP requests are mapped to the controller's methods. For instance, calls to the url `/customFacts/{factId}/params/{rank}/options` are mapped to the method `getParamOptions()`.

The URL in `@RequestParam` supports parameters. This pattern, called *URI templates*, provides a way to access parts of the URL. In this case, it has variables `factId` and `rank`.

Annotation `@PathVariable` on a method argument binds it to the value of a URI template variable, e.g.: `/customFacts/CRITICAL_STATE/params/0/options` maps parameter `factId` to `CRITICAL_STATE` and `rank` to `0`. Those variables are then used in calls to methods. A `@PathVariable` argument can be of any simple type such as `int`, `long`, `Date`, etc. Spring automatically converts to the appropriate type. If the conversion fails, it throws an exception `TypeMismatchException`.

`@ResponseBody` annotation is used to bind a function return value to the value of the HTTP response body. In the G-ACM, objects in the body of requests and responses are transmitted in *JSON* format. For instance, the following object is return by the call to URL `/customFacts/SEALED_RESOURCE/params/0/options`:

```
{
   "fact":"SEALED_RESOURCE",
   "rank":0,
   "options":[
      {  "value":"ehr_observation_lab",
         "text":"Lab result" },
      {  "value":"ehr_action_administration",
         "text":"Administration" }
   ]
}
```

To assure decoupling between application modules, the controllers do not include application logic besides converting internal objects to Data Transfer Objects(DTOs) and vice-versa. DTOs are simple objects that have no behaviour and are used to transfer data between applications. Figure 6.8 shows `Principal` and `Pca` objects and their counterparts DTOs `PrincipalDTO` and `PcaDTO`.

To convert between Java objects and *JSON*, the G-ACM uses Spring's message conversion support. This mechanism requires the inclusion of the Jackson libraries `databind`, `core` and `annotations` [42] to the project. It is activated through the following entry in Spring's configuration file:

```
<mvc:annotation-driven>
   <mvc:message-converters>
      <bean class="org.springframework.http.converter.StringHttpMessageConverter"/>
      <bean class="org.springframework.http.converter.json.MappingJackson2HttpMessageConverter"/>
   </mvc:message-converters>
</mvc:annotation-driven>
```

This configuration assures that these conversions are done automatically at all points

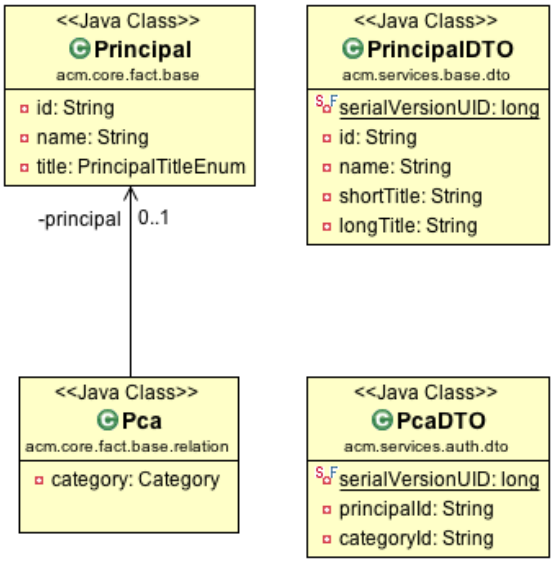

Figure 6.8: Classes Diagram: Data Transfer Objects

with annotation `@ResponseBody` or `@RequestBody`.

The complete list of services provided by the G-ACM Server is as follows:

- `BaseModelController`:
  - `/principals`: get the list of principals.
  - `/principals/{principalId}`: get principal with id `{principalId}`.
  - `/categories`: get the list of categories.
  - `/categories/{categoryId}`: get category with id `{categoryId}`.
  - `/actions`: get the list of actions.
  - `/actions/{actionId}`: get action with id `{actionId}`.
  - `/resources`: get the list of resources.
  - `/resources/{resourceId}`: get resource with id `{resourceId}`.
- `CustomFactsController`:
  - `/customFacts`: get the list of configured custom facts.
  - `/customFacts/{factId}/params/{index}/values`: get the list of acceptable values for the parameter at index `{index}` for custom fact with id `{factId}`.

- `AuthorizationController`:
  - `/pars` (POST): get the computed list of assignments principal/category ($\mathcal{PCA}$), assignments category/permission ($\mathcal{ARCA}$), assignments category/prohibition ($\mathcal{BARCA}$), and authorizations ($\mathcal{PAR}$ and $\mathcal{BAR}$) for the provided list of facts. As input, this service receives a list of `CustomFactInputDTO` containing the set of facts and their parameters values to be used when computing authorizations. Then, it uses the `CustomFactFactory`, described in paragraph **Custom facts configuration**, to build the corresponding custom fact objects from that input, and calls the method that processes the rules, described in **Rule processing**, above. Finally, it converts the results back to the corresponding DTOs and returns them to the caller.
- `ConfigController`:
  - `/configs`: get the list of available server configurations and acceptable list of values for each configurations.
  - `/configs` (POST): set configurations.

Server configuration This component derived from the need to configure the priority between authorizations and prohibitions in case of conflict, as described in Sections 4.2.2 and 5.2.5. Since the need for additional configurations was expected, it was decided to create a specific model, depicted in Figure 6.9, for this purpose.

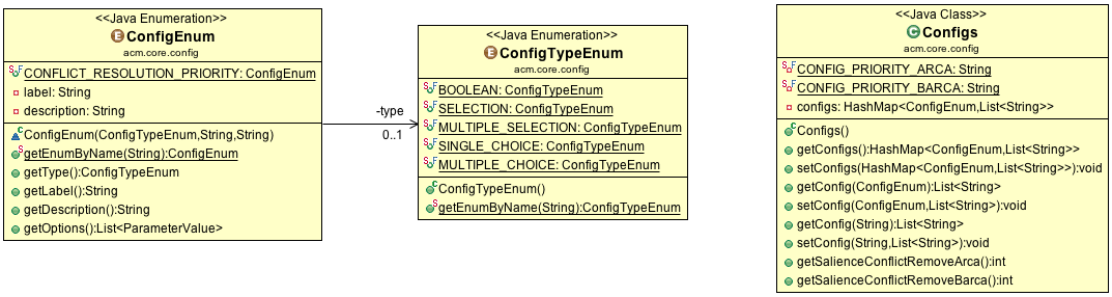

Figure 6.9: Classes Diagram: Config Model

Each value of `ConfigEnum` represents a configuration and has three attributes: `label` and `description` to explain the purpose of the configuration to users, and `type` that defines how the possible values for the configuration should be represented. Presently, `ConfigEnum` has only one entry:

```
CONFLICT_RESOLUTION_PRIORITY(
        ConfigTypeEnum.SINGLE_CHOICE,
        "Conflict priority",
        "Define if the system should give priority to
        authorizations or prohibitions in case of conflict.");
```

Type `ConfigTypeEnum.SINGLE_CHOICE` means that the values for this configuration should be displayed as a radio button. Additionally, `ConfigEnum` includes the method `getOptions` that returns the list of possible values for the configuration. For `SINGLE_CHOICE` it returns values `arca` and `barca` as described in the previous section.

Class `Configs` is initialized by the following Spring configuration:

```
<bean id="configs" class="acm.core.config.Configs">
  <property name="configs">
    <map>
        <entry key="CONFLICT_RESOLUTION_PRIORITY"
               value="${default.conflict.resolution.priority:barca}" />
    </map>
  </property>
</bean>
```

Note that the configuration's map is filled with the default values for the configuration. For `${default.conflict.resolution.priority:barca}` the value equals the value of property `default.conflict.resolution.priority`, but if that property is not set then `barca` is used.

6.4 Console

The G-ACM Console is developed in HTML and JavaScript and uses the Angular framework [37] to provide modularity and dependency management. D3 [38] is used to draw the graph. We start by describing through some examples how these tools are used in the G-ACM. Finally, we provide a brief description of the different modules and the way they interact with each other.

6.4.1 AngularJS

The main purpose of using Angular in the G-ACM is to provide a structure to support the application development and testing. Angular achieves this through a MVC philosophy to separate concerns and a DI mechanism. In this section we provide some examples of the use of the main features of Angular in the G-ACM.

Templates Templates are HTML documents with embedded Angular-specific elements and attributes. Angular combines the template with information from the model and controller to render the dynamic view that a user sees in the browser. The following code is part of the G-ACM's template for the dropdown used to choose custom facts:

```
<ng-form>
  <select ng-options=
    "cs as cs.getLabel() for cs in factsConfig.getFactsConfig()"
    ng-model="selectedFact">
  </select>
</ng-form>
```

Expressions beginning with `ng-` are Angular specific and are called *directives*.

Scope The scope corresponds to the application model, providing a context against which expressions are evaluated. Directives add watchers to the scope, which allow them to receive notifications on property changes.

Controllers and the view (or directives) do not have access to each other. This separation is important, since it simplifies the introduction of changes in the view and the controller, allows reusing the logic, and makes the code much easier to test. The scope provides the link between the controller and the view.

The scope is accessible to the special service `$scope` which detects changes to the model section and modifies HTML expressions in the view via a controller. The following excerpt shows this service being injected in the `MainController`:

```
controller('MainController', [ '$scope', 'Drawer', 'LayoutController', 'Configs',
    'SelectedConfigs', 'Sites', 'Principals', 'Categories', 'Actions', 'Resources',
    'Pcas', 'Arcas', 'Barcas', 'Permission', 'Pars', 'AcmServices', 'FactsConfig', 'Facts',
  function($scope, Drawer, LayoutController, Configs, SelectedConfigs,
    Sites, Principals, Categories, Actions, Resources, Pcas, Arcas, Barcas,
    Permission, Pars, AcmServices, FactsConfig, Facts) {
```

Controllers Controllers are JavaScript constructor functions used to add behaviour to the application by providing methods to manipulate the model (or scope).

The controller can add methods to the scope, which can be used by the view to react to events. For instance, the following excerpt from `mainScreenController`, adds to the scope the method `addSelectedFact`, which is used to add facts to the list of selected facts:

```
$scope.addSelectedFact = function () {
   /* Logic to store the selected fact */
}
```

In the template (`main.html`), the button to add facts calls that method when it is clicked:

```
<div class="parameter">
  <button  ng-click="addSelectedFact()"> Add </button>
</div>
```

Services and Dependency Injection Controllers should be simple, providing the logic for a single view. To reduce complexity and avoid code duplication, services should be used to share logic across controllers.

Services are registered by providing a service name and factory function in an Angular module. The following code registers the service `FactsConfig` used to hold the list of custom facts:

```
angular.module('AccessControlManager.entity', [])
.factory('FactsConfig',
  function(AccessControlServices, FactConfig, FactConfigParam) {
    var constructor = function () {
      // Constructor properties and methods
    }
    // Other methods
    return constructor;
  })
```

The service factory function creates a single instance of the service. All components that depend on the service receive a reference to that instance. The function returned by the service can be injected in any controller or service. Angular injector resolves the dependencies and instantiates components as needed.

There are several ways to annotate which services to inject in a function. The recommended way is the *Inline Array Annotation* that consists of passing an array with the service names, as well as parameters in the constructor function. For example, service `Principals` depends on services `Principal`, `Graph` and `Vertex`:

```
angular.module('acm.relation', [])
  .factory('Principals', ['Principal', 'Graph', 'Vertex', function(Principal, Graph, Vertex) {

     /* Service constructor, attributes and  methods */
    return constructor;
   }])
```

Injected services are now accessible in `Principals`' scope. For instance, method `getGraph()` instantiates service `Graph` and calls its method `setVertices()`:

```
    var graph = new Graph();
    graph.setVertices(this.getVertices());
```

Besides providing a way of organizing the code, Services and DI promote components decoupling, code reuse, and make it easy to replace and test individual components.

$http service The G-ACM uses `$http`, an Angular built in service, to communicate with the server. The `$http` service is a function that takes a single argument and returns a promise [43] with two methods: `success` and `error`.

`AccessControlServices` define a series of functions to call the services provided by the server. For instance, the following method calls the service that returns the available options for the provided fact and parameter:

```
accessControlServices.getValues = function(fact, param) {
    return $http({
      method: 'GET',
      url: urlBase + 'customFacts/' + fact + '/params/' + param + '/options'
    });
}
```

Since the returned value is a promise, it is possible to define the function to be called on service completion. The above service is used by the `mainScreenController` as follows:

```
AccessControlServices.getValues(factConfig.getType(), paramConfig.getRank())
.success(function (data, status, headers, config) {
      $scope.factsConfig.getFactConfigByType(factConfig.getType())
        .getParameter(paramConfig.getRank())
        .setOptions(data.options);
});
```

The provided function is called if the HTTP call is successful (response codes between 200 and 299). The parameter `data` is the object that represents the HTTP response body. In this example, no callback is provided in case the call fails.

The `$http` service provides default data transformations. In this case, it detects a *JSON* and automatically deserializes to the corresponding JavaScript objects. Service `$http` have additional features. For further reading we refer to [37].

Data binding This is the automatic synchronization of data between the model and view components. Data binding in Angular is bidirectional. Data changed in the model is immediately reflected in the view and vice versa. For instance, the following excerpt shows the code for the dropdown for selection of custom facts:

```
<select ng-options="cs as cs.getLabel() for cs in factsConfig.getFactsConfig()"
            ng-model="selectedFact"
            ng-change="initializeDefaults(selectedFact)">
</select>
```

The directive `ng-model` ensures that the variable `$scope.selectedFact` in the application model will have the current custom fact selected by the user.

DOM control structures DOM control structures are used for repeating, showing and hiding DOM fragments. The following code segment is responsible for displaying the list of facts already selected by the user:

```
<div ng-repeat="fact in selectedFacts.getFacts()" class="fact">
  <p>{{factsConfig.getFactConfigByType(fact.getType()).getLabel()}}</p>
  <div ng-repeat="param in fact.getParameters()" class="parameter">
    <div ng-repeat="value in param.getValues()" class="value">
      <p>{{ factsConfig.getFactConfigByType(fact.getType())
        .getParameter(param.getRank())
        .getOptionText(value) }}
      </p>
```

```
    </div>
</div>
```

The directive `ng-repeat` ensures that the inner segment will be repeated in the compiled view (or *live view*). The code above has three clauses `ng-repeat`: the first creates a `<div>` element for each selected fact; the second creates a `<div>` for each parameter; and the third creates a `<p>` to display the selected value for the parameter.

6.4.2 D3

The graph shown in the main screen of the G-ACM demands intensive manipulation of elements in the DOM. D3 is a Javascript library that provides functions to create, style and manipulate SVG objects. Next, we give a brief description of D3's main features. For the complete reference see [38].

Selectors D3 provides a declarative API based on selectors to simplify the task of performing transformations on DOM elements. For example, the code to change the text colour of paragraph elements in an imperative style is as follows:

```
var paragraphs = document.getElementsByTagName("p");
for (var i = 0; i < paragraphs.length; i++) {
  var paragraph = paragraphs.item(i);
  paragraph.style.setProperty("color", "white", null);
}
```

This can be replaced by the following code, using D3 selectors:

```
d3.selectAll("p").style("color", "white");
```

The Selectors API [44], is defined by the World Wide Web Consortium (W3C) and is supported natively by modern browsers, which make it fast even for large datasets.

D3 provides numerous methods for mutating DOM nodes: setting attributes or styles; registering event listeners; adding, removing or sorting nodes; and changing HTML or text content. These suffice for the vast majority of needs. Direct access to the underlying DOM is also possible, as each D3 selection is simply an array of nodes.

Dynamic Properties D3 allows specifying styles, attributes, and other properties as functions of data in D3, not just simple constants. For example, to alternate shades of gray for even and odd nodes:

```
d3.selectAll("p").style("color", function(d, i) {
  return i % 2 ? "#fff" : "#eee";
});
```

Computed properties often refer to bound data. Data is specified as an array of values, and each value is passed as the first argument (`d`) to selection functions. With the default join-by-index, the first element in the data array is passed to the first node in the selection, the second element to the second node, and so on. For example, an array of numbers can be bound to paragraph elements to compute dynamic font sizes:

```
d3.selectAll("p")
    .data([4, 8, 15, 16, 23, 42])
    .style("font-size", function(d) { return d + "px"; });
```

Enter and Exit Enter and exit selections are used to create new nodes for incoming data and remove outgoing nodes that are no longer needed.

When data is bound to a selection, each element in the data array is paired with the corresponding node in the selection. If there are fewer nodes than data, the extra data elements form the enter selection, which can be instantiated by appending to the enter selection. For example:

```
d3.select("body").selectAll("p")
    .data([4, 8, 15, 16, 23, 42])
  .enter().append("p")
    .text(function(d) { return "I'm number " + d + "!"; });
```

Updating nodes is the default selection of the `data` operator. Thus, when the enter and exit selections are omitted, the selected elements are those for which there exist corresponding data. A common pattern is to break the initial selection into three parts: the updating nodes to modify, the entering nodes to add, and the exiting nodes to remove.

```
// Update...
```

```
    var p = d3.select("body").selectAll("p")
        .data([4, 8, 15, 16, 23, 42])
        .text(String);

    // Enter...
    p.enter().append("p")
        .text(String);

    // Exit...
    p.exit().remove();
```

The operations run on the specified nodes, by handling these three cases separately. This improves performance and offers greater control over transitions.

6.4.3 Structure

This section describes the main components of the G-ACM Console, their interaction and the role that each one plays in the application.

The console follows the structure of Angular applications. File `index.html`, besides importing all needed JavaScript files, specifies the AngularJS module to load through the directive `ng-app`.

```
...
    <body ng-app="AccessControlManager">

        <ng-view></ng-view>
...
```

Directive `ng-view` simply includes the current rendered template in the indicated place. File `app.js` declares `AccessControlManager` as the main module of the application and defines the modules that the main module depends on:

```
angular.module('AccessControlManager', [
  'acm.drawer',
  'acm.entity',
  'acm.relation',
  'acm.graph',
  'acm.layout.controller',
  'acm.layout.data',
  'acm.services',
  'acm.controllers',
```

```
  'acm.config',
  'acm.fact',
  'acm.layout.tooltip',
  'ngRoute'
]).
(...)
```

Angular's `$routeProvider.when` method defines the G-ACM routes. Since we have a single screen it has only one entry:

```
config(['$routeProvider', function($routeProvider) {
  $routeProvider.
    when("/main", {templateUrl: "partials/main.html", controller: "MainController"}).
    otherwise({redirectTo: '/main'});
}]);
```

This code states that, when the URL includes the fragment `/main`, the application screen will be built from `partials/main.htm` template and `MainController`.

Figure 6.10 shows how the `MainController` relates to the remaining services and how these relate with each other.

Each module/service plays a specific role in the application:

- `MainController` orchestrates the various services used by the application.
- `AcmServices` contains all the methods used to contact with the G-ACM Server.
- Services in module `acm.entity` represent the base entities and relations.
- Module `acm.relation` contains the services that hold the list of base entities, base relations and authorizations. Relation services include methods to build graph objects from relations.
- Services `Vertex`, `Edge` and `Graph` are used to represent relations as graphs.
- Module `acm.fact` contains services to hold the custom facts configuration and the list of user selected facts.
- Services in `acm.config` hold the list of available configurations.
- `LayoutController` contains all the logic related to the graph creation and update (mostly using the D3 API).

- Service `LayoutData` builds the graph data from the graphs that represent the relations. It converts the provided graphs to the data that is actually used in the graphical representation. It includes methods to compute node positions in the screen, finds paths in the graph, etc.

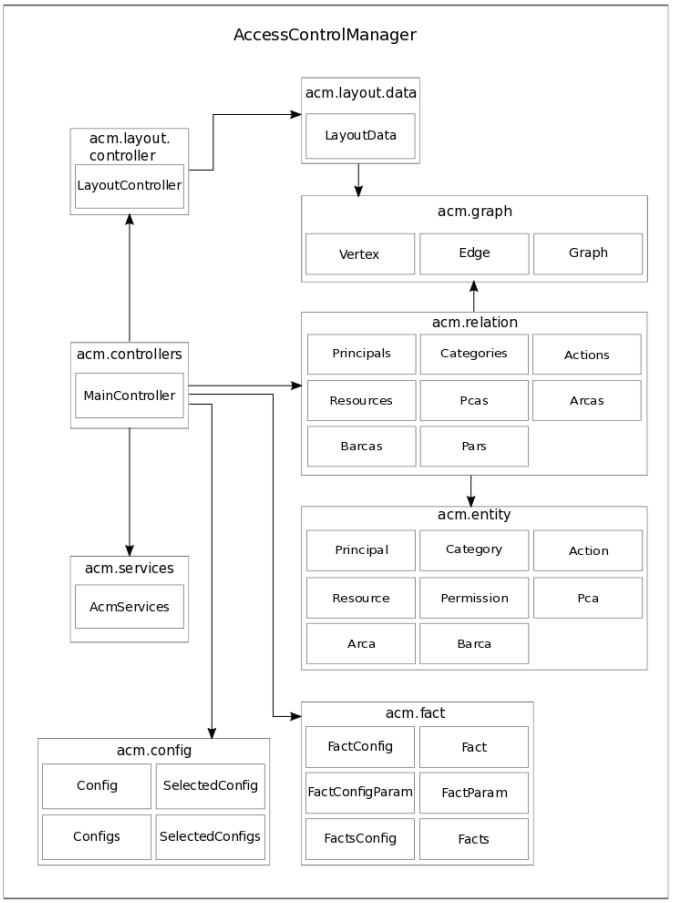

Figure 6.10: Console Controllers and Services

On application start up, `MainController` is loaded and the following sequence of events take place:

- `MainController` uses `AcmServices` to get the base entities, base relations and the list of authorizations from the server (when the service is called for the first time, no custom facts are provided, thus the set of authorizations returned is the one derived from the relations defined statically by configuration, as described in Section 6.3).
- `MainController` converts base entities and relations to `Graph` objects (`graph.js`) and creates an instance of `LayoutController` providing those graphs, the set of authorizations and the list of custom facts selected by the user (empty in the beginning).
- `LayoutController` creates an instance of `LayoutData` from the graphs and authorizations. `LayoutData` contains the actual data that is used to draw the graph. Besides data for entities, relations and authorizations, it includes visualization data, e.g. the nodes position in the display area, the flag that indicates if a node is selected, etc.

To simulate scenarios, the user should use the menu on the left pane. The button UPDATE invokes the authorization service providing the chosen facts and parameters. This triggers a sequence that updates the graph:

- `MainController` invokes the method `updateData` on `LayoutController` providing the new set of graphs, authorizations and the list of custom facts selected by the user.
- `LayoutController` stores the previous data in the history list and creates a new instance of `LayoutData` from the graphs and authorizations.
- `LayoutController` builds the graph from the new instance of `LayoutData`.

Chapter 7

Conclusion and Future Work

This work describes the implementation of a prototype, the G-ACM tool, for the visualization and analysis of access control policies following the CBAC model. The tool comprises a server to compute the permissions in the system and a user-friendly graphical console. The G-ACM Server uses the Drools rule engine to implement the dynamic aspects of the CBAC model. The system allows the configuration of custom facts, and rules that depend on them, to simulate specific access control requirements. The G-ACM Console provides a graphical representation of policies, which exposes the benefits of this representation for access control policy management.

The G-ACM tool allows the user to add or remove custom facts and change the values of their parameters. This provides an intuitive way of simulating environmental conditions, which is essential to understand how changes in the system state affect authorizations. It is also possible to compare graphs for two different simulation scenarios. Some additional features improve graph readability: types of edges in the graph (corresponding to CBAC relations) can be hidden/shown; nodes that have the same set of connections can be grouped, etc.

Security administrators query policies to extract information related to status of the system in terms of permissions and prohibitions, for instance, get the set of permissions for a certain principal, the mapping between principals and categories, etc. They also query policies about their correctness, in this case to verify if policies are well written. For instance, to look for categories without permissions or resources that are not accessible.

In [45], analysis queries are classified into *policy metadata queries*, *policy content queries* and *policy effect queries*. In [9] it is shown how a graph representation of policies can be used to answer to common queries of these types.

The G-ACM Console exemplifies, in practice, how such queries can be answered. It can be observed that the graph representation has some inherent advantages when compared to the traditional tabular representation.

Traditional RBAC systems represent the mapping between users and roles in one table and the mapping between roles and permissions in a different set of tables (one for each role). The graphical representation used by G-ACM, on the other hand, displays the full set of CBAC relations in a single graph. This provides a single view that displays the complete state of the system in terms of permissions. Naturally, for large numbers of entities, the ability of reading the graph also decreases but this can be compensated by a proper set of filters.

Figure 7.1 shows the RBAC plugin of the popular Jenkins open source continuous integration tool [46]). This application uses a variation of a role/permission table which, by adding additional columns, can represent the mappings for all profiles in the same table. This solution has its own drawbacks. In real scenarios it is likely that a given permission is only applicable to a small set of profiles. With this solution a column for each permission will always be represented whether it is applicable to the profile or not. Additionally, actions must be repeated for each resource that they are applicable to.

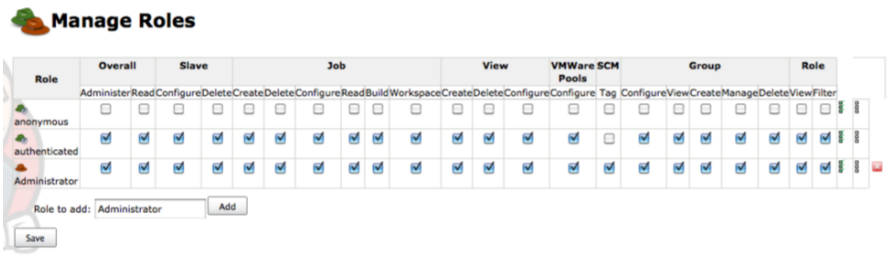

Figure 7.1: Jenkins RBAC plugin

It is easy to see that this solution does not scale for increasing number of resources and/or actions, the number of columns increases rapidly making it very difficult to read the permissions. Furthermore, the mapping between users and roles is provided by a different table making it difficult to verify the actual set of permissions for a given user.

In G-ACM, on the other hand, only the existing relations are represented and an action

is represented by a single node no matter how many resources it is mapped to.

Figure 7.2 shows an example of mapping between categories and permissions ($\mathcal{ARCA}$) in a tabular fashion. Figure 7.3 displays a snapshot of G-ACM for the same set of $\mathcal{ARCA}$ values. In the tabular form, the representation of the relation $\mathcal{ARCA}$ requires 105 cells, whether in G-ACM the same information is represented by 33 links.

		Specialist	Resident	Intern	Registered Nurse	Nurse Practitioner
Prescription	Read	V	V	V	V	V
	Create	V	V	X	X	X
	Cancel	V	V	X	X	X
Administration	Read	V	V	V	V	V
	Administer	V	V	V	V	V
	Cancel	V	V	X	X	X
Lab order	Read	V	V	V	V	V
	Create	V	V	X	X	X
	Cancel	V	V	X	X	X
Specimen collection	Read	V	V	V	V	V
	Perform	V	V	V	V	V
	Cancel	V	V	X	V	X
Lab result	Read	V	V	V	V	V
	Create	V	V	X	V	X
	Cancel	V	V	X	V	X
Image exam order	Read	V	V	V	V	V
	Create	V	V	X	V	X
	Cancel	V	V	X	V	X
Image exam result	Read	V	V	V	V	V
	Create	V	V	X	V	X
	Cancel	V	V	X	V	X

Figure 7.2: ARCA relation as table

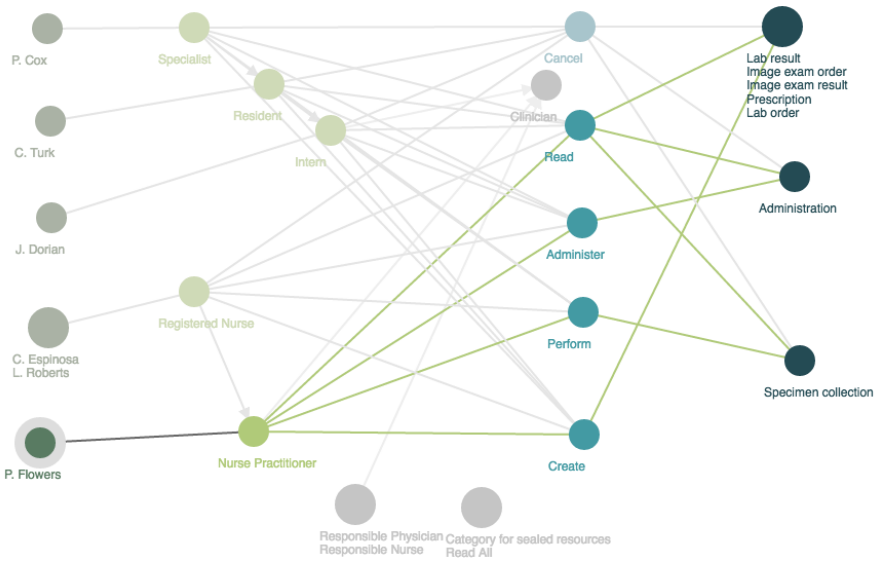

Figure 7.3: ACM snapshot

Even though, the graphical representation may look complex it is important to note that it shows much more information than the table in Figure 7.2. Besides the $\mathcal{ARCA}$ relation

it also shows the mapping between principals and categories ($\mathcal{PCA}$), the categories hierarchy, and the complete set of authorizations and prohibitions ($\mathcal{PAR}$ and $\mathcal{BAR}$). After overcoming the apparent complexity of the screen, a user can easily verify the set of permissions for a specific user simply by selecting the node corresponding to that user.

The representation used by G-ACM has some additional benefits. The ability of grouping the nodes that have the same set of neighbors can greatly simplify the graph and, therefore, its interpretation. This feature, which is not easily translatable to the tabular representation, is important because in real world scenarios it is common to have many principals assigned to the same categories.

Another important feature in G-ACM is the ability of simulating how the system state affects the permissions. Without the single view provided by the graphical representation this feature would be seriously limited since users would have to look at several different tables to verify how the permissions were affected by each change in the values of the parameters.

From the above, we consider that the exploration of the graphical representation of policies to help users with the tasks related to the management of security policies is a worthwhile endeavor. Nevertheless, more definitive conclusions will require a more extensive analysis and, to do so, several important features should be added to the G-ACM prototype.

The distributed version of the CBAC model, which would allow to configure more realistic scenarios where multiple policies, under the responsibility of different agents, are combined to provide a unified answer to access requests. The inclusion of obligations [47, 48, 49], would allow the definition of actions that the user or the system have to execute so that an authorization is conceded.

To improve G-ACM's usability, several additional developments can be considered. The node grouping feature can be improved by allowing the user to choose which nodes to group. The inclusion of filters to select the entities to show in the graph would increase its readability.

To give a better understanding on how the custom facts affect authorizations, the information about the facts responsible for the creation of edges should be visible. At the time of this writing that information is already returned by the server, but not yet displayed.

To explore new ways of displaying the graph, the D3's *force layout* is being used. This type of layout positions nodes through the simulation of physical forces. This feature is still under development but the first version can already be seen in the prototype [50].

The simulation of authorization scenarios in the G-ACM is intuitive. To use the G-ACM Console, a basic understanding of the CBAC model is enough. Editing rules, on the other hand, requires a high degree of technical knowledge. The CBAC Console should provide a simpler way of creating and editing the rules. Editing code directly, the approach used in the *Policy Manager*, has some weaknesses. It implies that the user must know how to write the rules, and introduces security risks. The chosen approach should define a language that simplifies the process of editing rules and, simultaneously, imposes limits on the actions that the rules can perform. An interesting possibility is to use the *Drools Workbench* [35], the rules authoring and management module of Drools, to provide a graphical interface for editing rules.

Bibliography

[1] Lujo Bauer and Florian Kerschbaum. What are the most important challenges for access control in new computing domains, such as mobile, cloud and cyber-physical systems? In *Proceedings of the 19th ACM Symposium on Access Control Models and Technologies*, SACMAT '14, pages 127–128, New York, NY, USA, 2014. ACM.

[2] David Elliott Bell and Leonard J. LaPadula. Secure computer systems: A mathematical model, volume II. *Journal of Computer Security*, 4(2/3):229–263, 1996.

[3] Butler W. Lampson. Protection. *Operating Systems Review*, 8(1):18–24, 1974.

[4] Ryan Ausanka-crues. Methods for access control: Advances and limitations, 2001.

[5] Richard Kuhn Ravi Sandhu, David Ferraiolo. The NIST Model for Role-Based Access Control: Towards A Unified Standard. In *Proceedings, 5th ACM Workshop on Role Based Access Control*, pages 47–63, 2000.

[6] Steve Barker. The Next 700 Access Control Models or a Unifying Meta-model? In *Proceedings of the 14th ACM Symposium on Access Control Models and Technologies*, SACMAT '09, pages 187–196, New York, NY, USA, 2009. ACM.

[7] C Bertolissi and M Fernandez. *Category-Based Authorisation Models: Operational Semantics and Expressive Power*, volume 5965 LNCS, pages 283 – 301. Springer-Verlag, 2010.

[8] Sandra Alves and Maribel Fernández. A Framework for the Analysis of Access Control Policies with Emergency Management. *Elsevier Science, B.V.*, 2014.

[9] Sandra Alves and Maribel Fernández. A Graph-Based Framework for the Analysis of Access Control Policies. 2015. submitted for publication. http://www.dcc.fc.up.pt/~sandra/papers/AF2015.pdf.

[10] Maribel Fernández Clara Bertolissi. Rewrite specifications of access control policies in distributed environments. In *Proceedings of the 6th international conference on Security and trust management*, STM'10. Springer-Verlag, 2011.

[11] Clara Bertolissi and Maribel Fernández. A metamodel of access control for distributed environments: Applications and properties. *Inf. Comput.*, 238:187–207, 2014.

[12] Asad Ali and Maribel Fernández. Hybrid enforcement of category-based access control. In *Security and Trust Management - 10th International Workshop, STM 2014, Wroclaw, Poland, September 10-11, 2014. Proceedings*, pages 178–182, 2014.

[13] Manuel Koch, Luigi V. Mancini, and Francesco Parisi-Presicce. A graph-based formalism for RBAC. *ACM Trans. Inf. Syst. Secur.*, 5(3):332–365, 2002.

[14] Manuel Koch, Luigi V. Mancini, and Francesco Parisi-Presicce. Graph-based specification of access control policies. *J. Comput. Syst. Sci.*, 71(1):1–33, 2005.

[15] James A. Hoagland. Specifying and Implementing Security Policies Using LaSCO, the Language for Security Constraints on Objects. *CoRR*, cs.CR/0003066, 2000.

[16] Allan Heydon, Mark W. Maimone, J. D. Tygar, Jeannette M. Wing, and Amy Moormann Zaremski. Miró: Visual Specification of Security. *IEEE Trans. Software Eng.*, 16(10):1185–1197, 1990.

[17] David E. Bell and Leonard J. Lapadula. Secure Computer System: Unified Exposition and MULTICS Interpretation. Technical Report ESD-TR-75-306, The MITRE Corporation, 1976.

[18] Steve Barker and Maribel Fernández. Term rewriting for access control. In *Data and Applications Security XX, 20th Annual IFIP WG 11.3 Working Conference on Data and Applications Security, Sophia Antipolis, France, July 31-August 2, 2006, Proceedings*, pages 179–193, 2006.

[19] A. Santana de Oliveira. *Réécriture et Modularité pour les Politiques de Sécurité*. PhD thesis, Université Henri Poincaré, Nancy, France, 2008.

[20] Claude Kirchner, Hélène Kirchner, and Anderson Santana de Oliveira. Analysis of rewrite-based access control policies. *Electr. Notes Theor. Comput. Sci.*, 234:55–75, 2009.

[21] Clara Bertolissi, Maribel Fernández, and Steve Barker. Dynamic Event-Based Access Control as Term Rewriting. In *Data and Applications Security XXI, 21st Annual IFIP WG 11.3 Working Conference on Data and Applications Security, Redondo Beach, CA, USA, July 8-11, 2007, Proceedings*, pages 195–210, 2007.

[22] Tony Bourdier, Horatiu Cirstea, Mathieu Jaume, and Hélène Kirchner. Formal specification and validation of security policies. In *Foundations and Practice of Security - 4th Canada-France MITACS Workshop, FPS 2011, Paris, France, May 12-13, 2011, Revised Selected Papers*, pages 148–163, 2011.

[23] Clara Bertolissi and Maribel Fernández. A rewriting framework for the composition of access control policies. In *Proceedings of the 10th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 15-17, 2008, Valencia, Spain*, pages 217–225, 2008.

[24] Clara Bertolissi and Worachet Uttha. Automated analysis of rule-based access control policies. In *Proceedings of the 7th Workshop on Programming languages meets program verification, PLPV 2013, Rome, Italy, January 22, 2013*, pages 47–56, 2013.

[25] Suroop Mohan Chandran and James B. D. Joshi. *LoT-RBAC*: A location and time-based RBAC model. In *Web Information Systems Engineering - WISE 2005, 6th International Conference on Web Information Systems Engineering, New York, NY, USA, November 20-22, 2005, Proceedings*, pages 361–375, 2005.

[26] James Joshi, Elisa Bertino, Usman Latif, and Arif Ghafoor. A generalized temporal role-based access control model. *IEEE Trans. Knowl. Data Eng.*, 17(1):4–23, 2005.

[27] Devdatta Kulkarni and Anand Tripathi. Context-aware role-based access control in pervasive computing systems. In *SACMAT 2008, 13th ACM Symposium on Access Control Models and Technologies, Estes Park, CO, USA, June 11-13, 2008, Proceedings*, pages 113–122, 2008.

[28] H. Mirzapour-Aghdaghi and M. Fernández. Policy Manager: a tool to analyse category-based access control policies. `http://policymanager.herokuapp.com`, 2014. [Online; accessed 24-August-2015].

[29] Fabien Autrel, Frédéric Cuppens, Nora Cuppens-Boulahia, and Céline Coma-Brebel. MotOrBAC 2: a security policy tool. In *SARSSI'08 : 3ème conférence sur la Sécurité des Architectures Réseaux et des Systèmes d'Information, 13-17 octobre, Loctudy, France*, 2008.

[30] Anas Abou El Kalam, Salem Benferhat, Alexandre Miège, Rania El Baida, Frédéric Cuppens, Claire Saurel, Philippe Balbiani, Yves Deswarte, and Gilles Trouessin. Organization based access contro. In *4th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY 2003), 4-6 June 2003, Lake Como, Italy*, page 120, 2003.

[31] NEMA/COCIR/JIRA Security and Privacy Committee (SPC). Break-Glass – An Approach to Granting Emergency Access to Healthcare Systems. Technical report, NEMA (National Electrical Manufacturers Association), 1300 North 17th Street, Suite 1847, Rosslyn, VA 22209 USA, December 2004.

[32] HIPAA. Break Glass Procedure: Granting Emergency Access to Critical ePHI Systems – HIPAA Security. `http://hipaa.yale.edu/security/break-glass-procedure-granting-emergency-access-critical-ephi-systems`, 2015. [Online; accessed 01-September-2015].

[33] Ana Margarida Ferreira, David W. Chadwick, Pedro Farinha, Ricardo João Cruz Correia, Gansen Zhao, Rui Chilro, and Luis Filipe Coelho Antunes. How to securely break into RBAC: the BTG-RBAC model. In *Twenty-Fifth Annual Computer Security Applications Conference, ACSAC 2009, Honolulu, Hawaii, 7-11 December 2009*, pages 23–31, 2009.

[34] Health and Patient Care Information Center. Patient choices. `http://systems.hscic.gov.uk/infogov/confidentiality/choices`, 2015. [Online; accessed 2-March-2015].

[35] Red Hat. Drools. `http://www.drools.org`, 2015. [Online; accessed 27-May-2015].

[36] Charles Forgy. Rete: A Fast Algorithm for the Many Patterns/Many Objects Match Problem. *Artif. Intell.*, 19(1):17–37, 1982.

[37] Google. AngularJS. `https://angularjs.org/`, 2015. [Online; accessed 01-June-2015].

[38] Mike Bostock. D3.js. `http://d3js.org/`, 2015. [Online; accessed 01-June-2015].

[39] Inc. Pivotal Software. Spring. https://spring.io/, 2015. [Online; accessed 01-June-2015].

[40] Inc. Pivotal Software. Spring MVC. http://docs.spring.io/spring/docs/4.2.0.RC1/spring-framework-reference/htmlsingle/#mvc, 2015. [Online; accessed 20-June-2015].

[41] The Apache Software Foundation. Apache Maven. https://maven.apache.org/, 2015. [Online; accessed 29-August-2015].

[42] FasterXML. Jackson. http://wiki.fasterxml.com/JacksonRelease20, 2015. [Online; accessed 20-June-2015].

[43] Mozilla Developer Network. Promise. https://developer.mozilla.org/pt-PT/docs/Web/JavaScript/Reference/Global_Objects/Promise, 2015. [Online; accessed 2-August-2015].

[44] W3C. Selectors. http://www.w3.org/TR/selectors-api/, 2015. [Online; accessed 14-June-2015].

[45] Elisa Bertino, Carolyn Brodie, Seraphin B. Calo, Lorrie Faith Cranor, Clare-Marie Karat, John Karat, Ninghui Li, Dan Lin, Jorge Lobo, Qun Ni, Prathima Rao, and Xiping Wang. Analysis of privacy and security policies. *IBM Journal of Research and Development*, 53(2):3, 2009.

[46] Jenkins. AngularJS. https://jenkins.io/index.html, 2015. [Online; accessed 01-Nov-2015].

[47] Gansen Zhao, David W. Chadwick, and Sassa Otenko. Obligations for role based access control. In *21st International Conference on Advanced Information Networking and Applications (AINA 2007), Workshops Proceedings, Volume 1, May 21-23, 2007, Niagara Falls, Canada*, pages 424–431, 2007.

[48] Dirk Jonscher. Extending access control with duties - realized by active mechanisms. In *Database Security, VI: Status and Prospects. Results of the IFIP WG 11.3 Workshop on Database Security, Vancouver, Canada, 19-21 August 1992*, pages 91–112, 1992.

[49] Naftaly H. Minsky and Abe Lockman. Ensuring integrity by adding obligations to privileges. In *Proceedings, 8th International Conference on Software Engineering, London, UK, August 28-30, 1985.*, pages 92–102, 1985.

[50] João Sá. G-ACM - Graphical Access Control Manager. http://acm-joaosa.rhcloud.com/app/, 2015. [Online; accessed 25-September-2015].